EXTENSIONS OF CHANNEL ROUTING TECHNIQUES AND THE IMPLEMENTATION OF A STANDARD-CELL PLACEMENT AND ROUTING SYSTEM

by

Douglas Braun

Memorandum No. UCB/ERL M85/77

19 September 1985

EXTENSIONS OF CHANNEL ROUTING TECHNIQUES AND THE IMPLEMENTATION OF A STANDARD-CELL PLACEMENT AND ROUTING SYSTEM

by

Douglas Braun

Memorandum No. UCB/ERL M85/77

19 September 1985

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Extensions of Channel Routing Techniques and the Implementation of a Standard-Cell Placement and Routing System

Douglas Braun

Electronics Research Laboratory
Electrical Engineering and Computer Science Department
University of California
Berkeley, California 94720

ABSTRACT

A high-performance standard cell placement and routing system has been developed. A major part of the system is the channel routing. Extensions to the channel router YACR have been made to make it usable in a wide range of practical applications. In addition, a new multi-layer channel router has been developed, based of some of the ideas of YACR.

September 16, 1985

Acknowledgements

I am very grateful to all the people here at Berkeley who made it possible for me to study here and get as much out of it as I did. I am especially grateful to my research advisor, Professor Alberto Sangiovanni-Vincentelli, for giving me the opportunity to get exposed to so many things. I had no idea how much I would learn when I started working for him. I also want to thank him for being so easy to get hold of, especially when I was working around the clock to get this report finished.

I am also grateful to Professors Richard Newton and John Ousterhout for exposing me to many things in CAD and Computer Science, and to Carlo Sequin, for thoughtful advice during the writing of this report. I must also thank my fellow graduate students of the CAD group for answering millions of questions, giving lots of advice, and providing both simulating and incredibly stupid discussions. Special thanks go to Tom Quarles, Peter Moore, and Richard Rudell for giving far more than their share to keep our group going.

Several people contributed work that made this project possible. The work of Jim Reed and Carl Sechen form a fundamental part of this project. I must especially thank Jim for his solid original implementation of YACR that has stood up to endless modifications, and Deirdre Ryan for her code that inspired Termite. Thanks also to Jeff Burns, Fabio Romeo, and Karti Mayaram for supplying the Chameleon partitioner. Karti, along with Srinivas Devadas and Tony Ma, did me a great favor by quickly making changes in their own programs to provide me with data to compare Chameleon with.

I must also thank all the people who helped me when I wasn't in the office: the folks at Mary Morse Hall for getting me off to a good start, Kong for showing me how easy I got off, Todd, Scott and Lori and everybody else in San Jose for inspiring me to apply here in the first place, and Elizabeth for not getting sick of me when I worked late. I am especially grateful to Bob and Ruth Sicular for their gracious hospitality during the final

crunch, which made it possible for me to finish this project without having to sleep on my desk.

This work was supported by SRC under grant number SRC-82-11-008, and the MICRO program of the state of California.

Table of Contents

CHAPTER 1

Introduction

As the complexity of integrated circuits increases, performing the layout of the circuit becomes an increasingly complicated part of the design, and as circuits become more complicated, more time must be spent arranging and interconnecting the components, or *cells*, of the circuit compared to designing the cells themselves. Also, the proportion of space occupied by the wiring on a chip increases as the circuit complexity grows. It is not uncommon to see VLSI chips where two-thirds of the area is occupied by interconnections instead of active circuitry.

The task of arranging the cells (*placement*), and the task of placing the connections between them (*routing*), affect each other. The goal of the placement is to arrange the cells of the circuit in the smallest possible area, and to minimize the length of the connections between them. Enough room must be left between the cells to hold the wiring, but it is very difficult to predict how much room will be needed. If the routing fails because of lack of room, it may be necessary to start the entire placement and routing process over again. To make the routing simpler, it is usually done in two stages, *global* and *detailed* routing. Global routing assigns a path for each net, but does not specify its exact location, while detailed routing places the wiring for each net in its final position. The detailed routing is usually performed by dividing the area between the cells into a series of rectangular regions called *channels*. This is because the detailed routing can be simplified by routing one channel at a time.

The most generalized placement and routing problem is the *custom-cell* problem. Rectilinear cells of arbitrary size with terminals at any point on their edges must be placed on the chip, and all the appropriate terminals connected. Custom-cell placement and routing is extremely difficult because the placement, global routing, and detailed routing strongly

interact with one another. The channels will have many common borders, and the order in which they are routed is significant. If one channel cannot be routed in the space allocated to it, the cells that form its borders will have to be moved further apart. This will change the configuration of the other channels adjacent to the cells, and they will also have to be rerouted. It is difficult to devise a scheme for defining the channels and their routing order that will guarantee not to propagate these errors indefinitely.

Standard-cell placement and routing is less difficult, but more restrictive. In this technique, the cells are required to be approximately the same size, and are placed in a series of parallel rows. When this is done, the channel definition is obvious, and the routing of each channel can be done almost independently. This makes it much easier to create an automated placement and routing system that will be able to guarantee to find a solution, and consistently produce high-quality solutions.

One of the goals of our CAD group here in the Department of Electrical Engineering and Computer Science at the University of Cal., Berkeley has been to produce an automated integrated circuit synthesis system. Such a system would ideally produce a finished layout directly from a logical specification of the circuit. As a first step, we are planning to create a system that will take a logical function described by a set of Boolean and state equations and produce a finished layout implementing the function. Such a *random logic synthesis* system would be able to implement more complicated functions than other automated techniques such as PLAs.

A standard-cell design style is a good way to implement this system for three reasons. First, the cells of the circuits, such as gates and registers, are relatively small and uniformly sized, making them well-suited to the standard-cell design style. Second, it is possible to create a system that will always be able to place and route the cells, and consistently produce high-quality results without manual intervention. Finally, tools already have been developed at Berkeley that do an excellent job at many of the necessary tasks. These include the channel router **YACR**, and the standard-cell placement and global rout-

ing programs in the **TimberWolf** package.

This report describes the work done towards automating placement and routing, especially for the standard-cell design style. The channel router YACR has had its capabilities extended to handle a greater range of routing problems. YACR has also formed the basis for a new multi-layer channel router. This router will give us the ability to take advantage of the larger number of layers that state-of-the-art IC fabrication technologies make available for interconnection. The enhanced version of YACR and TimberWolf have also been integrated into the standard-cell placement and routing system **ThunderBird**. This system is compatible with the design environment at Berkeley.

Chapter 2 describes two modifications done to YACR and incorporated into the official version of the program. One modification is the ability to specify precisely fixed terminals on one side of the channel. This allows YACR to route a region with fixed terminals on three sides, which has generally required a less efficient switchbox router. With a 3-sided channel router, it is possible to route L-shaped channels, which is desirable when routing a custom-cell chip. The other modification is the ability to route channels whose width is not uniform. Such channels often exist on a custom-cell chip, or can be formed when a standard-cell circuit is created with cells of different heights. This ability is exploited by ThunderBird.

Chapter 3 describes the multi-layer channel router **Chameleon**. Chameleon can route channels using any number of layers greater than one, and can respect design rules that dictate a different spacing and width of the material on each layer. Its fundamental algorithms are derived from YACR, with modifications to take advantage of the extra degrees of freedom that more than two layers provide. The results obtained have been excellent, with the advantage that the performance does not degrade when only two layers are used. As a two-layer channel router Chameleon's performance is equal to or better than that of YACR.

Chapter 4 describes the standard-cell placement and routing system ThunderBird. ThunderBird uses YACR for channel routing (with the modification for irregular channels mentioned above) and the TimberWolf placement program for the cell placement and global routing. It is designed to obtain the circuit and cell data from the Squid database management system, and produces a physical layout in the same format. Its modular nature allows it to use data from other design systems, and several industrial circuits, including a 2500 cell example, have been placed and routed with ThunderBird.

Chapter 5 gives the conclusions, and the Appendix contains documentation for the latest version of YACR, Chameleon, and ThunderBird.

CHAPTER 2

Extensions to the Channel Router YACR

2.1. Overview of Channel Routing

Channel routing [Ha71],[Yo82],[Sa84],[De76],[Ri82] has been used for a number of years in integrated circuit routing. It is especially useful because channel routers can consistently produce a good solution to a channel routing problem, and more complex routing problems can often be broken down into a number of channel routing problems.

A *channel* is a rectangular region which has terminals at regular intervals on two opposite sides, the *top* and *bottom* of the channel. It is also possible to have terminals on the left or right *ends* of the channel, but their exact location usually cannot be specified in advance.

Each terminal is to be connected to some subset of the others. Each terminal belongs to exactly one *net*, and all the terminals of any net are to be connected together. There is usually a *grid* superimposed over the channel, which divides the channel into a series of *rows* and *columns*. Terminals may be placed only where a column intersects the top or bottom of the channel, and pieces of material, or *segments*, that make the connections, may lie only along a row or column.

There are generally two layers of material available for making the connections. One layer (the *vertical layer*) is generally used for vertical segments, while the other layer (the *horizontal layer*) is used for horizontal segments. If a connection must be made between segments on different layers, a *contact* is placed at a row and column location that they have in common. The pattern of wiring for a particular net may consist of a single horizontal segment between the leftmost and rightmost terminals of its net, with vertical segments running up and down from the terminals to the horizontal segment. Sometimes,

several separate horizontal segments will be used for one net, with extra vertical segments connecting them. This technique is known as *doglegging* [De76]. Terminals at the ends of the channel are created by allowing a net's horizontal segment to extend out to the appropriate edge.

The input to the routing problem is the lists of nets that are to be connected to each position at the top and bottom of the channel, as well as the list of nets that are to have terminals on the left and right edges of the channel. The lengths of the top and bottom lists must be the same, and determine the number of columns, or *length* of the channel. If a position on the top or bottom of the channel does not have a terminal, this fact is noted by placing the *null net* at the appropriate position in the top or bottom lists. In practice, nets are identified by number, and the top and bottom lists are a series of integers, with zero representing the null net. It is meaningless for a zero to appear in the lists of nets exiting the ends of the channel.

While the length of the channel is specified by the input, its *width*, or the number of rows it has, is determined by the channel router. Minimizing the width of the channel is the primary measure of quality of a solution, while additional goals are minimizing the total wire length or the number of contacts.

The *local density* at a particular column of the channel is the number of nets that must cross the column, or have terminals at the top or bottom of the column. The *density* is the maximum value of the local density over every column. The density provides an excellent measure of the minimum possible width of the channel. If the density is N, there must be at least N horizontal segments at some column of the net. If all these segments are on the horizontal layer, N rows will be needed to hold them.

A *vertical constraint* exists when two nets have terminals in the same column. When this happens, the net connected to the top terminal must have its horizontal segment above that of the net connected to the bottom terminal. If this is not obeyed, a *vertical constraint violation*, or *VCV*, exists, and the two nets cannot be directly connected to their

terminals with simple vertical segments.

All the vertical constraints can be represented by a directed graph, the *vertical constraint graph*, or *VCG*, where the nodes represent nets, and the edges represent vertical constraints. A directed edge from node i to node j means that the horizontal segment of net i must be placed above the segment of net j. If the vertical constraint graph is cyclic, it is impossible to eliminate all VCVs when placing the horizontal segments of the nets. When this happens, doglegging of some form must be used to complete the connections.

2.2. Overview of YACR

YACR attacks the channel routing problem in two stages. First, a single horizontal segment for each net is placed on the layer reserved for horizontal wiring. The placement algorithm, guided by information derived from the vertical constraint graph, can always place these segments in an area whose width is equal to the density of the channel. Vertical segments are then placed on the other layer to connect each pin at the top or bottom of the channel to the horizontal segment of the pin's net. When a vertical constraint violation makes it impossible to place the vertical segment, a series of specialized maze routers are used to attempt to make the connection. If the maze routing succeeds, the routing is obviously completed with a channel width equal to the density of the channel. Otherwise, the routing is started over with a wider channel, which will leave more free space for the routers to exploit.

The algorithm for placing the horizontal segments is as follows. First, a column whose density is equal to the maximum density of the channel is chosen to be the *initial column*. From the set of all nets that cross this column, a net is selected using the *select* algorithm, and is placed in a row chosen by the *assign* algorithm. This is repeated until all the nets crossing the initial column have been placed.

The *Modified Left Edge* algorithm (or LEA), and *Modified Right Edge* algorithm are then used to place the rest of the nets. The LEA places the nets from the initial column to

the left of the channel, and its counterpart places nets from the initial column to the right end. Since the two algorithms are essentially mirror images, only the LEA will be described further.

The LEA is based on the Left Edge algorithm originally proposed by Hashimoto and Stevens [Ha71]. It examines each column one at a time, starting at the initial column and moving rightward. At each column, if there exists one, or at most two, nets whose left endpoint is in the column, they are added to the set of placeable nets. Then if there are any nets in the set of placeable nets whose right endpoint is in the column, nets are repeatedly selected from the set using the *select* algorithm, and placed in a row chosen by the *assign* algorithm, until the set is empty.

The above algorithm is guaranteed to place the horizontal segments of all the nets without any overlap in an area whose width is equal to the density of the channel, independent of the choice of the *select* and *assign* algorithms. However, no attention is paid to avoiding vertical constraint violations. The implementations of *select* and *assign* described below try to minimize the number of violations created.

Select and *assign* depend heavily on a cost function which is based on the level of the nets in the vertical constraint graph. Before any nets are placed, this graph is created. Since cycles in the graph make the concept of level meaningless, they are first removed by deleting edges which create them. This is done heuristically, attempting to select edges that correspond to columns in the channel which the maze routers will most likely be able to route.

The cost function is calculated for every net-row combination before any nets are placed. The level from the top of the graph and the level from the bottom determine for each net the highest and lowest rows in which the net can be placed without definitely causing a VCV. The rows outside this region are given a *high* cost, and the rows within the range (assuming its width is greater than zero) are given a *medium* cost. Several heuristics are used to adjust this cost function.

Select and *assign* use this cost function as follows. *Select* simply chooses the net in the set of placeable nets whose total cost, summed over all rows, is greatest. For a given net, *assign* first looks at all rows with a *medium* cost, and chooses the one which will cause the least number of VCVs with respect to the nets already placed. If there are no rows of *medium* cost, or all of them are blocked by other nets, then the entire channel width, except for the first and last rows, is examined. If this fails, all row positions are examined. The properties of the LEA guarantee that there will be at least one row in which the net can be placed.

As each net is placed, the vertical segments connecting it to its terminals are also placed. If a VCV prevents a vertical segment from being placed, the number of its column is added to the list of columns containing VCVs. After all the horizontal segments and all the vertical segments without VCVs are placed, a series of specialized maze routers are used to attempt to route the connections in the columns with VCVs. These maze routers may place a horizontal segment on the vertical layer if necessary to make the connection. If they fail to route one or more columns, YACR first attempts to insert an additional row somewhere in the channel which will allow the maze routers to complete the connections. If this fails, the entire routing process is started over with one more row added to the channel.

The robustness of YACR is largely due to the maze routing. Most notably, the presence of cycles in the vertical constraint graph is not a severe obstacle. Each cycle corresponds to a VCV that cannot be removed by any ordering of the nets in the channel. Since the LEA almost always ends up creating a number of VCVs even when no cycles are present, the addition of a few more due to cycles is not a great problem. The only potential problem is that the VCVs caused by cycles will always be present no matter how wide the channel is, while non-cyclic channels can always have their nets placed without any VCVs if the channel is at least as wide as the depth of the vertical constraint graph. Although it is very rare for a channel to fail to route using the above algorithms, an addi-

tion to YACR was made which would add extra columns to the end of the channel if it could not otherwise be routed. With this addition, YACR will always be guaranteed to route any channel.

2.2. Addition of Fixed End pins.

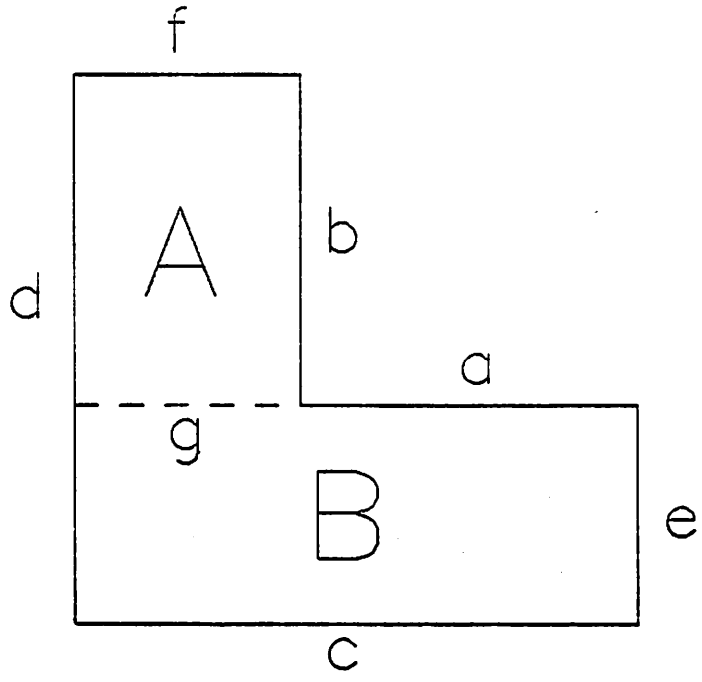
2.2.1. Motivation

One limitation of channel routers is that many IC design styles have more complicated routing situations than simple channels. A general custom-cell chip has a number of rectilinear blocks which can be of arbitrary sizes and locations, with the spaces between the blocks available for routing. To perform the routing of these chips, present-day methods are usually based on dividing the space between the blocks into rectangular routing regions, and attempting to route each region with a channel router. There are two problems that make this difficult. One is that the amount of space needed by the routing is not known until the routing is done. If more space is needed for a channel than is available, then the blocks must be moved to make room, which can force re-routing of many other channels. The other problem is that, because none of the end pins of a channel can be fixed before the channel is routed, there are constraints on the order in which the channels must be routed. If the graph of these constraints contains cycles, then it may be necessary to route a channel that has fixed pins on all four sides. This is the *switchbox* routing problem, which is much more difficult than the channel routing problem.

Defining and ordering the channels using the concept of *slicing structures* [Ot82] will produce channels without fixed end pins whose width can be adjusted without affecting other channels. However, an arbitrary placement of blocks will not necessarily allow the creation of a slicing structure; it may be necessary to adjust the positions of the blocks before the routing is attempted.

Dai, Asano, and Kuh [Da85] have proposed a system for defining and ordering routing regions that avoids the problems mentioned above. However, their method requires a

router that can route L-shaped channels. Such a channel has the two sides of a regular channel replaced by one concave and one convex 'L' shaped section.



An L-shaped Channel

The channel can be divided into two rectangular sections A and B. The channel routing problem for part A consists of the pins on segment b, and the pins on the portion of segment d that forms the boundary of section A. The floating end pins are the nets that pass through segment f (one end of the entire channel), and segment g (the common border of A and B).

After section A is routed with a normal channel router, the nets that pass between the two sections have had their positions fixed. Now the pins on three sides of section B (segment c, the portion of segment d that forms the border of section B, and the union of segment a and the border of sections A and B) are fixed. To route section B, a channel router that can handle fixed pins on one end of the channel is needed. YACR has been extended so that it can handle this situation.

2.3.2. Modifications to YACR's algorithms

The algorithm used by YACR first places nets in the channel starting near the center of the channel, and uses the LEA to sweep left and right, placing the nets at the ends of the channel last. Because there is no doglegging, if a pin has a fixed position on the end of the channel, its net must occupy the row that intersects the pin. In order to guarantee that each net can be placed without blockages, it is necessary to first place the nets that the fixed end pins belong to, and then perform the left (or right) edge algorithm starting at the end of the channel which has the fixed pins, moving to the other end of the channel. Because the LEA will succeed for any placement of the nets crossing the initial column (which is now the end column), it is clear that it will always be possible to place all the nets.

Even though all the nets can be placed in the channel, it is still necessary to deal with the vertical constraint violations that exist after the nets have been placed. YACR tries try to minimize the number of violations produced by starting the placement at the densest point of the channel, and placing the nets crossing this column in an order which will hopefully minimize the number of resulting violations.

When there are fixed end pins, the choices of initial column and initial net placement are predetermined. Therefore, there will generally be more vertical constraint violations with fixed end pins than without.

Another potential problem with fixed end pins is that any constraints between the nets of the end pins will always be present because of the fixed ordering of the nets, even if the number of tracks is increased to be greater than the channel density. These constraints are similar to those produced by cycles in the vertical constraint graph, which also cannot be eliminated by adding more tracks to the channel. Whether or not YACR's maze routers will be able to handle these violations without needing an excessive number of added tracks must be evaluated experimentally.

2.3.3. Implementation of fixed end pins

The previous version of YACR allowed the user to specify the order, but not the exact position of the pins at one end of the channel. The user interface was extended to allow precisely fixed end pins. See the manual pages in the Appendix for an exact description of the input file format.

One restriction is that the number of pins (or empty spaces) specified must exactly equal the number of rows of the channel. Since YACR normally starts with a number of rows equal to density, at least that many pins must be specified. If the number of pins specified is greater than the density, then the channel width is set to be equal to this value. Also, if routing cannot be completed in the initial number of tracks, then YACR gives up instead of adding a track. This is because YACR has no way of knowing where the new empty end pin corresponding to the new row should be placed in the list of fixed end pins.

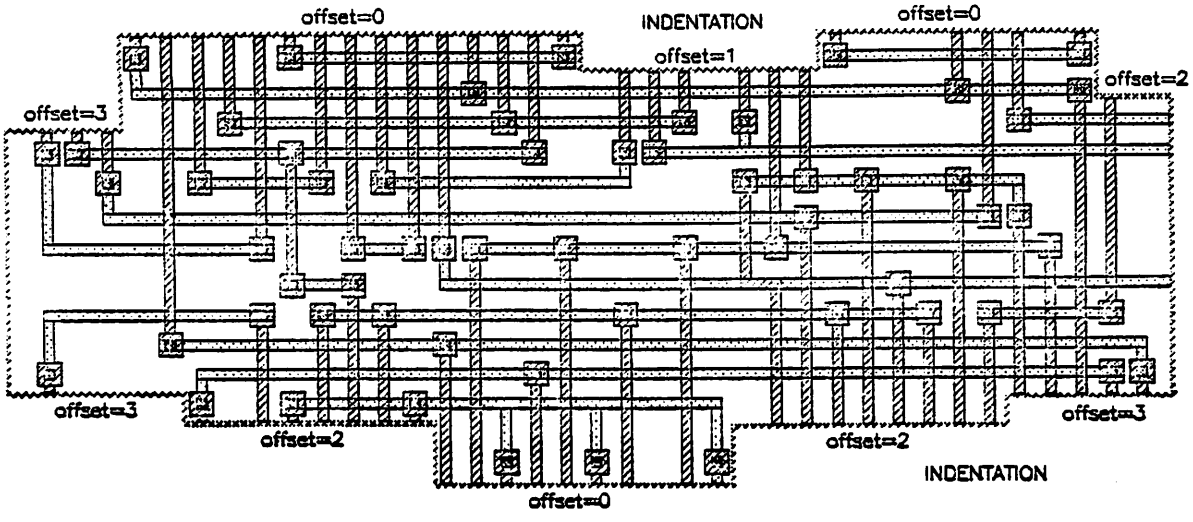
Considering how fixed end pins are used, these restrictions are not unreasonable. The configurations of the two L-shaped sides of an L-shaped channel are predetermined. If more room is needed when routing any part of the channel, an entire side may be moved, but the configuration of either side cannot be changed. If YACR fails to route one half of the channel, the act of making that half wider will also change the configuration of the other half, which will also have to be rerouted, which may in turn change the configuration of the first half.

2.4. Irregular channels

2.4.1. Definition and Applications of Irregular Channels

The top and bottom of a channel are normally each a single line segment. However, in certain applications, it is desirable to be able to route a channel where they consist of alternating horizontal and vertical line segments. Only the horizontal portions of the top and bottom would be allowed to contain pins. Each pin and unused pin position can then be considered to have an *offset*, which is its relative y coordinate, measured in grid units.

The pin(s) on the top and bottom furthest from the center of the channel have an offset of zero, and the value of the offsets increase the closer a pin is to the center of the channel.



An Irregular Channel

One application of irregular channels is in standard-cell integrated circuits. Usually all the standard cells in a certain block are designed with the same height. This allows the sides of the routing channels (created by the facing sides of two adjacent blocks) to be straight lines. However, imposing a fixed height on all cells may make the layouts of very simple or complicated cells awkward. Small cells may have much unused space, while large ones may be so long that they are difficult to efficiently pack into rows. Without a channel router that can handle irregular channels, there would be no overall reduction in space if a cell is shorter than the highest one in its row, because the indentations created by having a cell shorter (or taller) than its neighbors would not contain any horizontal portions of nets.

It is easy to see that such a channel can be routed by any channel router capable of handling normal channels. Simply move each pin whose offset is less than the greatest offset of any pin on that side towards the center of the channel, until all the pins of each

side are at the same y coordinate (and in a straight line). Route this channel using a normal channel router, then place vertical segments from the location of each pin back up (or down) to its original position.

The problem with this method is that it does not take advantage of the space within the indentations of the channel. An indentation is a region of the channel that has as part of its left or right boundaries one of the vertical segments that makes up a side of the channel. Modifications were done to YACR to allow the user to specify the offsets of each pin (thus implicitly defining the line segments that make up the sides and the indentations), and to allow YACR to take advantage of the space in the indentations when placing the nets.

2.4.2. Implementation of Irregular Channels

The changes to YACR necessary to implement irregular channels were:

1: Enabling the user to specify the offsets of the pins.

2: Modifying the density calculations and the cost functions for the placement of nets to reflect the fact that a net's horizontal segment cannot intersect an indented edge of a channel.

3: Modifying the maze routing functions to prevent them from trying to use regions above or below an indented part of the channel.

The offsets of the pins are specified by the user as two lists similar to the lists specifying the top and bottom pins. Each value represents the relative distance towards the center of the channel of the offset of each column. The absolute values are unimportant, as the lists are internally scaled to range from zero upwards.

In YACR, the contents of the channel are internally represented by two two-dimensional arrays, one for each layer. Each element is the number of the net whose wiring occupies that cell, or zero to represent an unused area. If the channel is irregular, the height of the array is equal to the maximum width of the channel. Therefore,

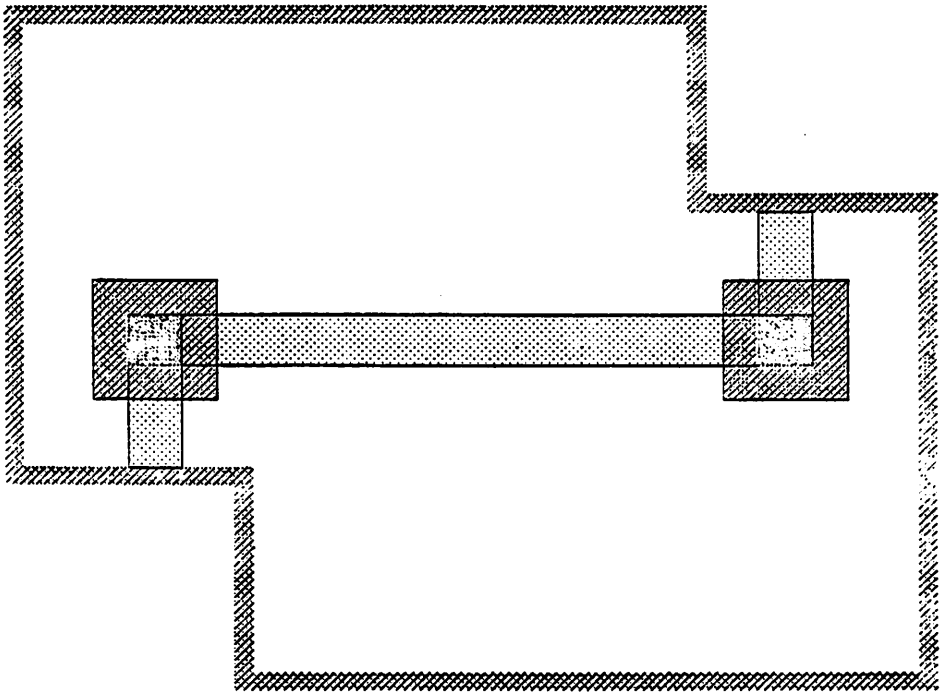
indentations correspond to portions of the channel in which it is illegal to place any wiring. Each grid cell that represents an indentation contains a non-zero value different than all the nets' numbers. This prevents nets from being placed in indentations, and maze routers from trying to use the indentations. The cost function that gives the cost of placing a net in a particular row was adjusted to make the cost of a row blocked by indentations to be infinite. The calculation of the density of each column was also modified. The density of a column is now considered to be the number of nets crossing the column, plus the sum of the top and bottom offsets of the column.

The calculation of the levels of the nets in the vertical constraint graph is also affected. Before leveling the graph, each net is given initial levels from the top and bottom of the graph equal to the number of rows into the channel the net must be placed to avoid intersecting an indentation. This ensures that the final levels of each net truly represent the highest or lowest column that a net can be placed in without causing vertical constraint violations.

The calculation of the density described above actually represents an absolute lower bound on the width of the channel. Because each net must use a single horizontal segment (no doglegs), it may be necessary to add more rows.

Instead of trying to calculate the minimum width that the modified left-edge algorithm will need to place all the nets, an attempt is made using the number of rows equal to the density. If a net cannot be placed, another row is added, and the placement is tried again

Other implementations of channel routers for irregular channels, such as the adaptation of Yoshimura and Kuh's router [Yo82] by Liu and Chen [Li81] have approached the problem by treating each indentation as a separate routing region, routing the indentations starting at the outermost ones, and finally routing the inner portion of the channel. Because no doglegs are used, our technique treats the channel as a whole. Also, if there are no indentations, the algorithms reduce to the exact same ones used by YACR for non-



Channel that Cannot be Routed in Density

indented channels. If no offsets are specified, offset lists of all zeros are used: the same program handles indented and non-indented channels without compromising the performance for non-indented channels.

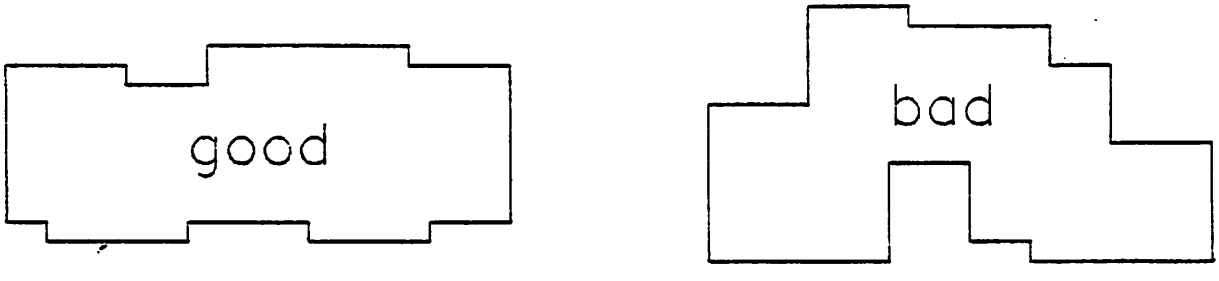
The techniques used by YACR to deal with indentations in the channel could also be used to add obstacle avoidance. An obstacle can be thought of as a rectilinear 'island' in the middle of the channel, or as an indentation that has broken off from the side of the channel. The density of the channel at any column would be the sum of the number of nets crossing the column, the top and bottom offsets, and the height of the obstacle at that column. The leveling of the vertical constraint graph would be affected as follows. If a net's level from the top of the graph is less than the distance from the top of the obstacle to the top of the channel, the net could fit above the obstacle without definitely causing any VCVs. Otherwise, the net would have to run below the obstacle, and its depth in the graph (and the depth of all nets below it) would be increased by the height of the obstacle. The level from the bottom of the graph is affected in the same way. One new con-

straint is that an obstacle cannot lie between a net's horizontal segment and one of its pins. This would be reflected in the cost function by setting the illegal rows to INFINITY.

2.4.3. Limitations of Current Approach

Our goal in implementing the ability to route irregular channels was to be able to give good results for channels with non-drastic indentations, while still preserving the excellent performance of YACR on non-indented channels. An important application of this version of YACR is as part of the automatic standard-cell placement and routing system ThunderBird. In this environment, there tend to be a relatively large number of small indentations in the channels, instead of a few deep ones. Also, the router must be robust, not failing to produce good results in the presence of 'real-world' obstacles like cycles in the constraint graph, or very deep constraint graphs.

The performance of YACR suffers if the indentations are too large, or are in unfavorable positions relative to one another. This is particularly evident if opposing indentations cause the channel to have a 'zig-zag' appearance. Because doglegs are not allowed, long nets cannot bend to get around the indentations.



Good and Bad Channels

A version of multi-layer channel router Chameleon (described in chapter 3) has the ability to place doglegs in nets; however it does not yet have the ability to handle irregular

channels. The modifications necessary to Chameleon to allow it to handle indentations are straightforward: all the modifications to YACR described above apply to Chameleon, because Chameleon is based on the same basic algorithms for net placement as YACR.

Allowing doglegs would also improve the obstacle avoidance capability mentioned above. Without doglegs, a bad combination of obstacles could prevent some nets from ever being placed. This situation would exist if a net had an obstacle under one of its top pins and another obstacle over one of its bottom pins, and the obstacles had a row in common. Without doglegs, there would be no column in which the net could be legally placed.

2.4.4. Experimental Results for Irregular Channels

The ability to handle irregular channels has been incorporated into the production version of YACR. A slightly different implementation has been incorporated into the production version of the TOPAGEN placement and routing system at Siemens in Germany. The new version of YACR has been tested with a large number of test cases which have straight edges (no offsets). As expected, in every case the results produced have been identical to those of the previous version.

Below are results for several test cases from Liu and Chen's paper [Li81] on their channel router for irregular channels. Listed are the dimensions of the channels, and the results obtained by YACR and Liu and Chen's router. For YACR's results, the minimum possible number of rows (taking into consideration the fact that doglegging is not possible) is given, along with the actual number of rows used, the total length of the wiring, and the number of vias required. For Liu and Chen's results, the rows used are given, along with the wire length and via count when known.

				YACR's Results				Liu and Chen's Results		
Example	Columns	Nets	Density	Minimum Rows	Actual Rows	Wire Length	Vias	Actual Rows	Wire Length	Vias
chen20	37	21	13	14	14	669	62	14	680	65
chen21	45	30	18	21	21	1227	65	20		77
chen22	79	54	21	21	22	2331	126	21		
chen23	37	21	22	22	22	887	61	22	891	75
chen24	61	47	25	26	27	2159	107	25		130

Results of YACR with Irregular Channels

For 3 out of the 5 cases, YACR was able to complete the routing using the minimum possible number of rows. Because of its ability to use doglegs, Liu and Chen's router was often able to complete the routing using less tracks than YACR.

Because Liu and Chen's router is based on Yoshimura and Kuh's channel router [Yo82], they should give approximately the same results for channels without indentations. The paper describing YACR [Sa84] gives a comparison of Yoshimura and Kuh's router and YACR for several test cases.

Because YACR does not use doglegs, it required fewer vias than Liu and Chen's router.

CHAPTER 3

Chameleon: A Multi-Layer Channel Router

3.1. Overview

The channel routing problem assumes that there are two layers of material available for routing. This is because until recently, IC fabrication processes had two layers, metal and polysilicon, available for wiring. Now, most processes that ICs are being designed for have three layers available for routing, and four layers are being used experimentally. Also, other electronic circuit fabrication technologies such as PCBs and hybrid circuits can have many layers of material available for interconnection. With these applications in mind, we have been working on extending the channel routing problem to handle more than two layers. Other progress in this direction has either been limited to 3- or 4-layer problems [Ch84], or has placed constraints on the arrangement of the wiring [Ha85]. Our goal has been to handle the problem in the most general way.

The basic concepts of the channel routing problem remain mostly unchanged. It is assumed that each layer is treated as a grid in the same way as before. One new specification of the problem is that the actual connections to the pins at the edges of the channel can be made using any of the layers. Also, the principle of two-layer channel routing that one layer is reserved for horizontal wiring, and the other for vertical, is no longer used (YACR already violates it anyway). In practice, it is often necessary to add additional constraints to the problem to satisfy design rules.

The concept of density is greatly affected by having more than two layers. The definition of density assumes that the horizontal portions of the nets are all on one layer. If there are more than two layers, the number of layers which carry horizontal segments may vary. For example, two routers may both claim to route a problem 'in density' for

three layers. But if one router uses only a single layer for horizontal segments, and the other two layers for vertical segments, while the other router can devote two layers to horizontal segments, the second router may route the channel in an area only half as large as the first. If the layers do not have the same pitch, the situation is even more ambiguous. Although the density of a channel is still a valid concept, it no longer directly indicates how wide the channel will have to be.

The algorithms used by YACR and other two-layer channel routers are relatively simple and effective because the use of two layers (the minimum possible to handle the channel routing problem) restricts the solution space that must be examined. When more layers are used there are many more possible solutions, and concepts that are very useful for solving the two layer problem such as the vertical constraint graph, can no longer be used in the same way.

Design rules cause other complications to the problem. The two layer problem assumes that the spacing of the grid is the same for each layer. This is usually not much of a problem in practice because the fact that one layer has mostly horizontal wiring, and the other mostly vertical, allows the grids to be deformed to fit the actual design rules. Also, the pitches for metal and poly in most single-layer metal processes are close to one another. When more layers are used, these assumptions fail. If each of several layers may contain both horizontal and vertical wiring, it is very difficult to transform the grids for each layer independently. Also, fabrication processes with more than two layers often have large variations in the pitch of different layers.

One approach to solving the multi-layer problem is to divide it into several two-layer problems, each of which deals with a portion of the nets, and uses two adjacent layers. This is effective, but it does ignore many possible valid solutions. Also, design rules such as the ones restricting the placement of contacts above one another can keep the two-layer sub-problems from being truly independent. If three layers are available (a very common case in practice), or any odd number of layers, this approach does not tell how to take

advantage of the odd layer.

The multi-layer channel router **Chameleon** has been written based on algorithms used by YACR. Its approach is to create two- or three-layer subproblems to aid the placement of the horizontal net segments, and then to treat all layers equally when using maze routing to complete the connections. It allows the user to specify the relative pitches of all layers, and it can enforce design rules prohibiting the stacking of contacts. The results obtained have been excellent. When only two layers are used, Chameleon can often outperform YACR.

3.2. Chameleon's Strategy

There are three basic parts to the strategy of Chameleon. First, the layers are divided into *groups*, and the nets are *partitioned* amongst the groups. The horizontal segments of the nets are then placed, using the same basic algorithms as YACR, and maze routing is used to connect the pins at the top and bottom of the channel to the horizontal segments of their nets.

As mentioned above, the multi-layer routing problem can be divided into a number of two-layer problems, with each of the layer pairs responsible for routing a subset of the nets. Chameleon's strategy is loosely based on this. It tries to place the wiring of a net on a particular subset of the layers, but it will violate this paradigm whenever necessary to produce a solution. The layers are usually treated identically by the algorithms, with the differences between them expressed as variations of cost functions.

3.2.1. Groups

The first stage of the algorithm is to divide the layers into *groups* of two or three adjacent layers. Each group contains one *vertical layer* and one or two *horizontal layers*. A *horizontal layer* primarily contains the horizontal segments of nets, while a *vertical layer* primarily contains the vertical segments connecting the horizontal segments to the pins of the channel. Horizontal wiring can be placed on a vertical layer (and vice versa), but only

if necessary to complete a route. If there are three layers, the vertical layer is between the horizontal layers.

Some other definitions that must be made are for *rows* and *tracks*. A row always corresponds to a particular y coordinate of the grid used to represent the routing, and refers to all layers at once. A track is a row on a particular layer that can hold a horizontal segment of a net. When several layers are considered at once, there are several tracks for each row.

A group with two layers is clearly analogous to the standard channel routing problem. When there are three layers, there is still a great deal of similarity. The horizontal segments of the nets are placed on the outer two layers, while the inner layer (which contacts both other layers) is used to make the vertical connections. About the only significant difference between the two-layer case and three-layer case is in vertical constraints. A VCV exists in the two-layer case if the horizontal segment of the net connected to the upper pin of a column is lower than the horizontal segment of the net connected to the lower pin. While this definition is the same for three layers, the problem is more difficult because each row of the channel can now contain two nets (one on each horizontal layer). Specifically, the depth of the vertical constraint graph that can be accommodated without any VCVs is only half of what it was before. This is because in the two-layer case, N tracks would have occupied N rows, in the three-layer case, N tracks occupy $N/2$ rows. It is actually the number of rows, not the number of tracks, that determines the depth of the vertical constraint graph that can be accommodated without VCVs. The result in practice is that the algorithms of YACR can be easily extended to three layers, but with a large increase in the number of VCVs that exist after the horizontal segments are placed.

When there are two or three layers, there is only one group. When there are more layers, the groups are created according to the following table:

No. of Layers	Arrangement
2	HV
3	HVH
4	HV HV
5	HVH VH
6	HVH HVH
7	HVH VH VH
8	HVH HVH VH
9	HVH HVH HVH
10	HVH HVH VH VH
...	...

Two things are significant about this arrangement. First, as many of the groups as possible have three layers. In fact, in every case the percentage of horizontal layers cannot be increased without leaving some horizontal layer without an adjacent vertical layer. As the number of layers approaches infinity, two thirds of the layers are horizontal layers. Also, the ordering of layers in the groups is arranged so that each horizontal layer has vertical layers on both sides as often as possible. Although this would not matter if each group was routed separately, it gives Chameleon more freedom to make connections.

The arrangement of horizontal and vertical layers determines the lower bound on the width of the channel. The determining principle is the same as for two layers; the number of tracks reserved for horizontal segments can be no less than the density of the channel. If the spacing between the tracks of each layer is uniform, the minimum width of the channel is simply the density divided by the number of horizontal layers (rounded up). If the spacing of the layers is different, the calculation is more complicated. The minimum width is most easily defined as the smallest width that contains the required number of tracks. In practice, Chameleon iterates starting with a width of one and counting the number of available tracks, until it finds the minimum channel width.

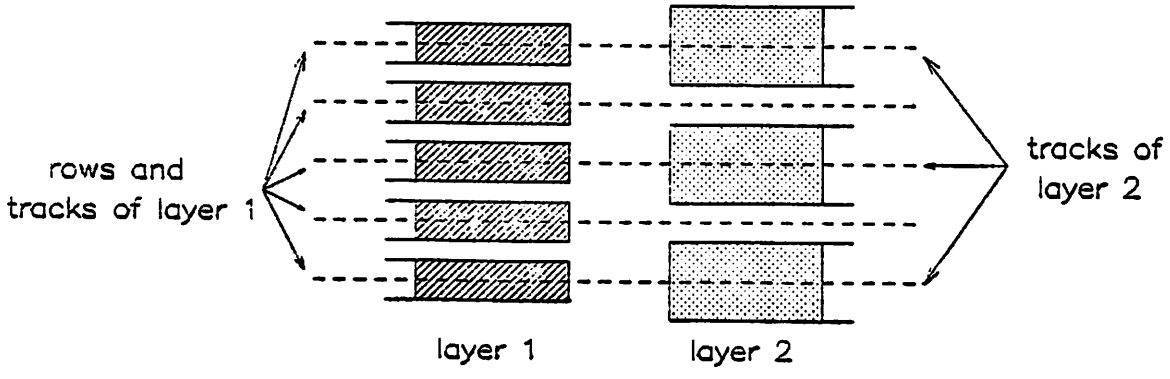
3.2.2. Variable Pitch of Layers

Chameleon can handle situations where the track-to-track spacing, or *pitch* of the layers is not uniform. The data defining the pitch of each layer (in terms of the width and separation of the material of each layer) and the number of layers used is read in from a

file.

The differences in pitch are ignored in a horizontal direction. Each column of the grid defining the channel is assumed to be capable of holding a vertical segment of any layer. Since the goal of the channel router is to minimize the width of the channel, while the length of the channel is fixed, this is not a problem. In the vertical direction, each grid unit represents an actual distance equal to the greatest common denominator of the pitches of the layers. For example, assume that one layer has a pitch of one, while the another has a pitch of three. A horizontal segment of the first layer may be placed on every horizontal grid line, while only every third grid line can hold a horizontal segment of the other layer. Note that the width of the material is not represented. Each segment actually represents the centerline of the material forming it. As long as the distance between segments of differents nets is at least as large as the pitch, no spacing violation will occur.

If the pitch of a layer is greater than one, the definitions of rows and tracks above have more implications. It is now the case that, when a single layer is considered, there are several rows for each track, instead of one row for each track. The difference between rows and tracks is important, and should be kept in mind.



Rows and Tracks for Layers With Different Spacing

When the horizontal segments are placed for the second layer described above (with a

pitch of three), every third row can hold a segment, while the rows between them can have no material placed on them. Such a rigid strategy would be unacceptable if applied to the wiring on the vertical layers. A fixed subset of the horizontal grid lines cannot be designated for wiring because the wiring on the vertical layers must be connected to the wiring on the adjacent horizontal layers, whose pitch may be unrelated to to the pitch of the vertical layer. Since two pieces of wiring on different layers must directly above one another to place a contact between them, a different strategy is used by the maze router (which is the only part of the program that places wiring on the vertical layers). When deciding if it is legal to place material in a particular grid cell, the maze router checks to see if the material would be too close to the material above and below it. It examines a number of cells above and below the cell of interest equal to the pitch of the layer minus one, and if any of them contain material not belonging to the net currently being routed, the cell of interest is rejected by the router. This approach guarantees that the spacing rules will always be satisfied, while allowing the maximum freedom in performing the maze routing.

3.2.3. Partitioning

After the layers are placed into groups, the nets are partitioned between the various groups. The assignment of a net to a group is not absolute. A cost function gives a relatively low cost to placing a net in its correct group, and a much higher cost to placing it on any layer outside of its group.

The way the nets are partitioned is critical to the success of the routing, because a vertical constraint between two nets disappears if the nets are placed on layers not belonging to the same group. In fact, the two nets can use different layers for their vertical connections. The effect this has on the vertical constraint graph is to remove all edges which connect nets belonging to different groups.

The goal of the partitioning is to assign the nets to groups in a way that allows all the nets to actually fit on the horizontal layers allocated to each group, and to eliminate

the vertical constraints that would be most likely to cause problems when the nets are actually placed.

The first condition may not always be satisfied. For example, if a channel routing problem has a density of ten, the two problems obtained by randomly dividing its nets into two groups might have a total density of more than ten. Although a partition can always be found which 'conserves density' (just take an already routed channel, and divide the nets by splitting the channel lengthwise down the middle), such a partition would be unlikely to minimize the number of VCVs. Chameleon will not fail if the nets in a group cannot actually be placed on the horizontal layers of the group, because the nature of the cost function will always allow the net to be placed somewhere.

To minimize the number of VCVs, we must try to do several things to the vertical constraint graph when assigning the nets to groups. First, we want to remove edges so that the depth of the vertical constraint graph is minimized. Also, we want to eliminate as many cycles as possible. Even if there are only two groups, the depth of the vertical constraint graph can be greatly reduced. For example, consider a graph that consists of a single linear path from one net to the next. If the nets are alternately assigned between one group and the other in the order that they appear on the path, every edge of the graph will be broken, and every vertical constraint will disappear. Similarly, if only one net forming a cycle in the graph is assigned to a different group than any of the other nets in the cycle, the cycle will disappear. When the partition is well done, Chameleon will almost always be able to route a multi-layer channel without having to add extra rows.

3.2.4. The Implementation of the Partitioning Algorithm

The actual partitioning of the nets is done by a separate program that is automatically executed by Chameleon. The same program also decides how the layers are to be divided up into horizontal and vertical layers and groups. The partitioning was implemented in this form so that it could be used as a preprocessor for other multi-layer routing programs developed at Berkeley. The actual algorithms for the partitioning were

developed and the program was written by Jeff Burns, Karti Mayaram, and Fabio Romeo [Bu85].

The current implementation of the partitioning program is optimized to work with a different multi-layer channel router. This channel router must treat channels with cycles in their vertical constraint graph specially, so the partitioner may produce a different arrangement of layers into groups than described above if cycles are present.

The partitioning algorithm used is as follows: First the composition of the groups is decided upon. Then each net is taken (in an arbitrary order), and the cost of placing it in each of the partitions is evaluated according to the following cost function:

$$cost = lp + d + 100*k*NC + 10*MAX(0, d-D) + 10*MAX(0, lp-LP)$$

where

lp	=	current longest path in the VCG of the sub-problem,
d	=	current density of the sub-problem,
k	=	0 if the sub-problem is of type VHV or VHVH,
k	=	1 if the sub-problem is of type HVH,
k	=	2 if the sub-problem is of type HV,
NC	=	number of cycles created in the VCG,
D	=	the ideal density of the sub-problem,
LP	=	bound on the depth of the vertical constraint graph of the sub-problem.

The net is placed in the least costly partition. After each net is placed, the vertical constraint graph of the nets already placed is regenerated and leveled. The cost function indicates that a very high cost results if a cycle is created by placing a net in a particular partition. A smaller cost results if placing the net in a partition would cause the density to exceed the number of tracks in the partition, or if the depth of the vertical constraint graph would exceed the number of tracks. Although this is a relatively simple algorithm, experience has shown that more sophisticated (and time-consuming) algorithms produce little improvement.

If the pitches of the various layers are non-uniform, the partitioning algorithm is more sophisticated. It takes into consideration the width and separation of the material forming each layer when calculating the maximum depth of the vertical constraint graph

that the group can accommodate. For example, if the width of the material of one layer of a group is wide enough to overlap several tracks of the other layer, the layer with the wider pitch actually reduces the depth of the vertical constraint graph that can be accommodated by the group. Another optimization done is the interchanging of the two layers of a HV partition so that the layer with the smallest pitch becomes the horizontal layer, maximizing the number of tracks of the partition.

3.2.5. The Vertical Constraint Graph

Once the nets have been partitioned, the vertical constraint graph is constructed, and then leveled (and cycles broken, if necessary). The algorithms and code to do this was used unchanged from YACR, and will not be described further. After the construction of the graph, but before the leveling and cycle breaking, the edges connecting nets in different groups are removed, because a vertical constraint effectively disappears when the nets belong to different groups. This has the effect of breaking the vertical constraint graph into a number of disjoint graphs, one for each group.

3.2.6. The Cost Function

The cost function that determines the cost of placing a net on a given layer and row is based on the cost function used by YACR. In fact, if two layers are used, each with a pitch of one, the values of the cost function will be identical to those of YACR, and the nets will be placed in the same positions.

The cost function is implemented as a 3-dimensional matrix $M(N, l, r)$ which stores the cost of placing net N in row r of layer l. The values of the function are calculated as follows: First, all rows on the vertical layers receive a cost of INFINITY. This enforces the rule that the horizontal net segments must be placed on horizontal layers. For each horizontal layer, the rows that do not correspond to a valid track position also receive a cost of INFINITY. For example, if a layer has a pitch of three, a horizontal segment could be legally placed in rows 1, 4, 7, 10, etc., without violating the spacing rules. Rows 2, 3,

5, 6, 8, 9, etc. would have a cost of INFINITY. Since the partitioning algorithm does not guarantee that every net of a group will actually fit on the horizontal layers of the group, the remaining rows of the horizontal layers not belonging to the net's group receive a cost of (2*HIGH). This allows them to be used, but only if there is no space to place the net on a layer belonging to its own group.

The remaining steps are very similar to the cost function of YACR. First the level from the top and the level from the bottom of the vertical constraint graph is calculated for each net. Because of the partitioning, these values will hopefully be much smaller than if the nets were not partitioned. These values are used to determine the highest and lowest rows that a net can be placed in without definitely causing a VCV, just as in YACR. The rows above and below this range are given a cost of HIGH. The rows within this range receive a LOW to MEDIUM cost, based on heuristics that try to minimize the chances of producing VCVs.

Because there may be two layers in a net's group (possibly with different pitch), that the net can be placed on, the calculation is made more complicated. If the pitch of the layers is uniform, the presence of two layers is easy to deal with. All that must be remembered is that the level of the net in the vertical constraint graph determines the number of *rows*, not the number of *tracks* that must be above or below the net when it is placed in the channel. If the pitches of the two horizontal layers are not the same, the number of rows is not clearly defined, because the tracks of the two layers can overlap. In this case, the layer with the wider pitch is ignored, and the layer with smaller pitch is used to determine how many rows must be above or below the net.

3.2.7. Placement of the Horizontal Segments

Once the cost function has been calculated, the horizontal segments of the nets are placed in the channel. From this point on, the distinctions between the groups are forgotten, and all layers are treated identically. The modified Left Edge algorithm used by YACR is used virtually unchanged. The *assign* and *select* procedures used by this

algorithm (described in section 2) are also almost unchanged. The main difference is that the assign function considers each row of each horizontal layer when deciding where to place a net.

A complication for the assignment algorithm is the calculation of the number of VCVs that would result if a net were to be placed in a particular row. In the two-layer case, it is easy to tell if a VCV exists in a particular column. A VCV exists if the net of the top pin of a column has its horizontal net segment on a row lower than the horizontal segment of the net of the lower pin. When there are more than two layers, more checking must be done. If the nets belong to different groups, there cannot be a VCV, because each net has its own vertical layer to use for the connection to the pin. If two nets belong to the same group, or even on the same layer, they may still be able to use separate vertical layers for the vertical connections. If the nets are on the same layer, this will be true if the horizontal layer has vertical layers both above and below it. It does not matter that one of the vertical layers does not belong to the group of the nets, because no more than two nets can have pins in any one column.

It is clearly difficult to tell ahead of time if the maze router will be able to route a column apparently containing a VCV without using any material outside of the column. In any case, if a mistake is made when deciding whether or not a VCV really exists, only the optimality, and not the correctness of the placement will be affected.

3.3. Chameleon's Maze Routing

3.3.1. Basic Approach

Once the horizontal segments of the nets are placed by the Left Edge and Right Edge algorithms, the vertical connections from the horizontal segments to the pins must be made. Most of the time (when the column containing the connection does not have a VCV), the connections are trivial. If there is a VCV in a column, maze routing must be used to find connections from the nets to the pins.

The maze routers used by YACR are specialized. They do not guarantee to find a path if one exists, but instead try several styles of paths that are most likely to be successful. With only two layers available they usually are successful. Also, they run quickly because they do not do an exhaustive search.

If there are more than two layers, it is very difficult to extend YACR's maze routers to exploit them. The code to do so would be very cumbersome and hard to change, and would have to be heavily dependent on the exact number of layers used, and how they are divided into horizontal and vertical layers. The addition of variable layer pitch and contact-stacking avoidance would only make things more difficult.

Because of this, I decided to write a more general-purpose maze router that would be more exhaustive in finding a path. Such a router would be simpler, and not be as greatly affected by the problems described above. The penalty paid for the increase in generality is a loss in speed, so it would be necessary to try to minimize the run time.

The biggest problem when using maze routers is that the connections made for one net can block nets that have not been placed. This problem is not too severe in our situation, because almost all of the connections have been already made before the maze router is used. Even so, care had to be taken to make sure that 'difficult' channels with many VCVs (often in adjacent columns) could be routed as close as possible to density.

3.3.2. The Maze Routing Problem

The basic problem for the maze router can be stated as follows: The channel is treated as a 3-dimensional matrix. Any cell C_n in the matrix is specified by an ordered triple (l_n, r_n, c_n), where l_n is the layer of the cell, r_n the row, and c_n the column. Each cell can contain material belonging to some net, or be empty. The input to the maze router is the cell C_i, referred to as the *initial cell*, and a *goal row* r_g and *goal column* c_g. The output of the maze router (assuming a result exists), is a list P of cells, each adjacent to the next. Two cells are adjacent if exactly one of $|l_i - l_j|$, $|r_i - r_j|$, $|c_i - c_j|$ is equal to

one, and the rest are zero. The first cell in the list is C_i, and the last, C_g, has its row equal to r_g and column equal to c_g. None of the cells in P may contain any material not belonging to the net being routed. The cells in P describe a path from C_i to C_g.

When routing a connection in the channel, C_i is the point on the net's horizontal segment in the same column as the pin we are trying to connect to. (Note that c_i is equal to c_g). r_g is either the top or bottom row of the channel, and c_g is the column containing the pin.

When searching for a path, the maze router finds one that minimizes a particular *cost function* $F_c(l, r, c)$, summed over all the cells in P. The cost function assigns a cost to each cell in the matrix. This cost may be static, or may depend on the locations of the adjacent cells in the list P. The exact characteristics of the cost function greatly affect the quality of the routing, and are described fully below.

At this point, a few parts of the cost function are of interest. These are the only parts that are parameters to the maze routing routine. They allow a higher-level strategy (described below) to adjust the operation of the maze router for any particular column. All other parts of the cost function are built into the routine itself. The first are the *left limit* L_l and *right limit* L_r. These specify the number of columns to the left and right of c_g that may contain the solution P. In other words, any cell C_j in P must have $c_g - L_l \leq c_j \leq c_g + L_r$. The purpose of these limits is to keep the routing out of areas where it might block other routing, and to speed up the running time of the router by keeping it from examining cells that could not possibly form part of the solution. The other parameters are the *soft left limit* S_l, and *soft right limit* S_r. They are similar to the left and right limits, except that they assign a very high cost to any cells outside the range they specify. This has the effect of allowing the solution to use cells outside the range only if a solution cannot otherwise be found.

3.3.3. High-Level strategy of maze routing

The first strategy tried was to first place all the connections that could be made without VCVs, and then apply the maze router to each column, allowing it to use every free part of the channel to make a connection. This caused the routing for relatively difficult columns to block other columns that had not been routed. The strategy that was finally used was to do the routing in several stages. The first stage is applied to all the VCV columns, and gives up if a connection is not found within its constraints. Then the second stage is applied to the columns that the first stage could not route. It tries harder to make a connection, reducing the constraints on a solution. The columns that it could not route are given to stage three, and so on until it is clear that there is no possible way to route a column. The various stages are essentially implemented as adjustments of L_l, L_r, S_l and S_r. Therefore, the code to implement a particular stage is quite simple.

Each of the stages must route two connections in every VCV column it is applied to. One connection goes from the top pin to its net, while the other goes from the bottom pin to its net, which is located above the upper pin's net. It is clear that one of the segments can always be placed directly from the net to the pin, while the other will have to jog around the first in some fashion. Sometimes it is necessary to force both nets to jog in order to route the column. Therefore, every stage consists of two applications of the maze router. The first routes one connection, and will always be able to find a solution if none of the routing for other columns has blocked the column of interest. The seconds attempts to make the connection to the other terminal. Because of the VCV, the routing cannot be in a straight line, and it will have to jog around the first connection somehow. When one of the stages attempts to route a column, first the top pin is routed, then the bottom pin. If both connections cannot be made, an another attempt is made starting with the bottom pin, and then the top pin.

3.3.4. The Various Stages

Stage one is applied to all columns, and it routes all the non-VCV columns. It will usually fail if there is a VCV in the column (always if only two layers exist). It sets L_l and L_r to a value of zero, forcing the routing to lie only in the column itself. This strategy routes the columns without VCVs as quickly as possible.

Stage two forces the first net routed to lie within the column (L_l and L_r equal to zero), while it allows the other net to use either of the two adjacent columns (L_l and L_r equal to one). This handles most VCVs while examining the smallest possible area, and is similar to the routine **mazela** in YACR, which also tries to use the adjacent columns to make the connection.

Stage 3 places the first net in a straight line, and then allows the other net to use a very wide region if necessary to complete the route. (L_l and L_r are set to twice the number of rows in the channel, because experience has shown that it is very rarely necessary to use a wider area). This is similar to the routine **maze2** in YACR.

Stage 4 tries to jog both nets. It places the first net with L_l equal to zero and L_r a large value. The soft left limit S_l is set to -1 This has the effect of forcing the router to choose the path as much as possible to the right of the column, essentially pushing the net over to the right as much as possible. The other net is then routed with left and right limits of infinity. Hopefully, it will be able to use the space in the column not used by the first net.

Stage 5 is a mirror image of stage 4, with left and right reversed.

A more sophisticated strategy was attempted as stage 6. First the routing for the columns adjacent to the unrouted column are ripped up, and are re-routed with soft left and right limits that tend to push the routing to the sides as far as possible from the unrouted column. Then all the previous stages are used to try to re-route the column. This strategy is effective in that it sometimes routes columns that could not otherwise be routed, but under certain circumstances, as when the routing for several pins of the same

net share the same material, it is difficult to avoid ripping up material that is needed for some other connection. In practice, this stage is not used when there is a chance of this situation occurring. Also, the verification routine will detect if any wiring is damaged. None of our test cases has had a verification failure with the current version of Chameleon.

The following table gives the number of times that the various stages succeeded for a number of test cases, including a set of especially difficult benchmarks, and a collection of channels from standard-cell chips. The relatively fast stages 2 and 3 succeeded in routing almost all of the VCV columns, while stages 4 and 5 and 6 were rarely needed. Also, it can be seen that the number of violations, and the number that cannot be routed without adding a row to the channel, decrease when more layers are used.

	2 Layers		4 Layers
Stage	Benchmarks	Standard-Cell Channels	Benchmarks
2	124	224	76
3	143	100	9
4	21	1	1
5	0	0	0
6	0	0	0
Failure	152	18	4

Table 3.1: Relative Usage of Maze Routing Stages

3.3.5. The Low-Level Maze Routing Algorithm.

The maze router used is based of the classical wave algorithm of Lee [Le61]. This algorithm operates by causing a 3-dimensional wave to propagate from the initial cell outwards in all directions. Obstructions absorb and diffract the wave, as if it were a wave of sound, or a wave on the surface of a body of water. As soon as a point on the wave front reaches the goal, the path that the wave took is traced backwards to the source to generate the path for the routing. If the wave is totally absorbed without ever reaching the goal, there cannot be a path to the goal.

The wave algorithm is more precisely defined as follows. Each cell C in the channel has a *direction pointer* associated with it. This pointer can point to any one of the six cells

adjacent to C, or it can be null. There is also a set of cells W called the *wavefront list*. Each cell in the wavefront list has associated with it a direction pointer as defined above, and a *cost* c. The algorithm runs as follows:

Set the direction pointers of each cell to null.

Make W empty.

Place the initial cell C_j in W, with its direction pointer set to null, and a cost of 0.

While W is non-empty:

Find the cell $C_j \in W$ whose cost is minimum.

If C_j is a goal cell, (i.e. $r_j = r_g$ and $c_j = c_g$), we have reached the goal. Begin the traceback procedure.

Otherwise, set the value of the direction pointer of C_j in the channel to the value of the pointer that C_j had in W.

For each of the six cells adjacent to C_j:

Evaluate the *cost function* $F(C_k)$. If the adjacent cell C_k is available for routing, the cost will be finite.

If the cost is finite, place C_k in W if it is not already there, along with its cost.

Set the direction pointer of C_k in W to point to C_j.

If C_k is already in W, update its direction pointer and cost only if the new cost is lower than the previous cost.

The traceback procedure is done as follows:

Make the list of cells P, the path from the initial cell to the goal, empty.

Add to P the cell C_g that was the last one examined above.

Add to the front of P the cell C_{g-1} that the direction pointer of C_g points to.

Continue this way, adding the cell C_j pointed to by the direction pointer of cell C_{j+1}, until C_i, the initial cell, is placed at the head of P.

P now contains the path from C_i to the goal.

As long as the cost function is non-negative, this procedure will produce the least costly path to the goal, if such a path exists.

3.3.6. The Cost Functions for Maze Routing

The nature of the cost function determines the characteristics of the routing. The present cost function has been chosen to produce routing that is least likely to block unrouted columns, while at the same time using the least amount of material, and minimizing the number of vias produced.

The cost function $F(C_k)$ mentioned above is actually calculated incrementally. That is, the cost of C_k is equal to the cost of C_j plus the cost of moving from C_j to C_k. (That is, $F(C_k) = F(C_j) + \tilde{F}(C_j, C_k)$.) The parameters of F indicate that the incremental cost of reaching a cell depends upon where the cell is, as well as where we are reaching it from. This second term is what allows the router to discriminate between horizontal and vertical wiring.

The cost function is evaluated as follows:

Any cell that cannot be occupied, because it is either outside the channel, already filled by another net's wiring, or outside the left and right limits L_l and L_r passed to the maze routing routine, has a cost of INFINITY, and cannot be further considered. The cell is also rejected if the cell's direction pointer in the channel is non-null. This means that the cell has previously formed part of the wavefront, and has already been examined. Other conditions that cause a cell to be rejected are if placing material in the cell would violate the spacing rule for the cell's layer, or if the cell lies in a column that has an unrouted VCV. The latter condition is not necessary to prevent illegal routing, but to ensure that routing of other columns will be more likely to succeed.

If the cell is located in a row that is not between the initial row and the goal row, it is also rejected. This keeps the router from wasting time looking in the wrong direction.

If the cell has not been rejected, its basic cost is equal to the Manhattan distance of the cell from the goal. The fact that the basic cost is not constant improves the performance of the router, as described below. If the column of the cell is outside the area defined by the soft limits S_l and S_r, a cost of HIGH (implemented as a value of 10000) is added. This gives the router a strong preference to avoiding any cells outside the soft limits unless a path cannot otherwise be found.

If the cell does not already contain wiring belonging to the net being routed, additional costs are added. This causes the router to try to re-use previously existing wiring. These costs are an additional unit cost, plus a cost of MEDIUM (implemented as a value of 100) if the cell is on a vertical layer that does not belong to the group that the net being routed is in. This causes the router to try to place vertical segments on the 'correct' vertical layer for the net.

More costs are added based on the current cell's predecessor C_j. If the router is attempting to place wiring in a vertical direction on a horizontal layer, or in a vertical direction on a horizontal layer, a cost of MEDIUM is added. This helps make it easier to route other columns, because a piece of wiring in the 'wrong' direction can block a number of tracks or columns. If C_k is on a different layer than C_j, a contact is necessary to connect the two layers. If such a contact cannot be placed because contact stacking would result, the cost becomes INFINITY, and the cell C_k is rejected.

The cost function moving from cell C_j to cell C_k can be summarized as follows:

$$F(C_k) = F(C_j) + L + d + 100000 * s + u * (1 + 100 * x)$$

where

L = 0 if C_k can be legally used, or INFINITY otherwise,
d = Manhattan distance from C_k to the goal,
s = 1 if C_k is outside the soft limits, 0 otherwise,
u = 1 if C_k is already occupied by the net's wiring,
or 0 if it has not been previously used,
x = 1 if we are moving horizontally on a vertical layer,
or vertically on a horizontal layer,
or 0 otherwise.

3.3.7. Tradeoffs in the Maze Routing

The various parts of the cost function that do not simply serve to reject illegal cells are heuristic; they have been experimentally adjusted to get the best quality routing for all channels that must be routed, without requiring any manual adjustment to route any particular channel. Chameleon must deal with a wider range of routing conditions than a traditional channel router. A traditional channel router only deals with two layers with the same track spacing, while Chameleon must be able to give quality results for any number of layers greater than one, and for large differences in the track spacing of the layers. As the cost functions and routing strategies were developed, results were often obtained for some channels that were better than the results obtained with the current version. In each case however, the overall performance of the router on a large set of test cases was inferior. From the results of many test cases, information can be obtained to improve the cost function and routing strategy that will allow the router to improve the results of certain cases, while not penalizing the overall performance.

One situation that would benefit from more sophistication in the maze routing is the difference between two-layer routing and routing with four or more layers. In the current implementation of Chameleon, the maze router is not adjusted in any way based on the number of layers. This is because one goal of this project was to devise maze routing algorithms that were equally applicable to any number of layers. In practice, if there are several layers (and consequently fewer rows), a net's horizontal segment will tend to be closer to the top or bottom of the channel. In this case, the rule against placing vertical

wiring on horizontal layers can often be relaxed. However, if this rule is removed completely, performance in the two-layer and three-layer case is greatly reduced.

Some parts of the cost function are more important for some of the higher-level strategies than others. For example, the rule that tries to keep vertical wiring on the vertical layer belonging to the net's group is really only necessary for stage one, whose only purpose is to route the trivial columns without VCVs. In fact, stage one could be replaced with specialized code to directly place the vertical segments. This was not done because the code for stage 1 is less than ten lines long, and works correctly as is.

Another reason for keeping the router simple is that special-purpose rules are difficult to maintain and bug-prone. For example, the strategy of stage 6 (described above) which requires the ripping up of previously routed columns cannot be used if a new row has been inserted somewhere in the channel in an attempt to complete the route. This is because the data that describes the routing results of each column is now no longer correct. The code necessary to fix this would substantially complicate the router, while only offering minimal benefit.

Overall, the heuristics chosen represent a good compromise between getting excellent performance without excessive time, and having simple, maintainable, robust code.

3.3.8. Speed Considerations

Generalized maze routing as used by Chameleon is relatively slow. Chameleon spends the majority of its time executing the maze routing code, while the specialized routers of YACR account for a relatively small portion of YACR's execution time. The next section contains an execution profile of Chameleon, showing the time spent in the different parts of the program, including the maze routing. The reason for the slow speed is that the router must examine every legal cell before it can conclude that a path to the goal does not exist. If a path does exist, the router might have examined every cell whose distance from the initial cell is less than that of the goal. These reasons suggest two

methods to reduce the number of cells examined. The first is to avoid looking at cells that are extremely unlikely to be part of a solution. The other is to try to expand the wavefront as much as possible in a direction towards the goal by choosing a cell to examine from the wavefront list F that is the most likely to be on the path from the origin to the goal.

The first technique is implemented in Chameleon's maze router by the left and right limits L_l and L_r. In the earlier stages of the routing strategy, these limits greatly increase the speed of the router by preventing it from wasting its time looking at the entire channel when it is not necessary. Another technique used is only allowing the router to search in a region between the origin and goal. If the goal is a pin at the top of the channel, the router will never look below the origin. We have never observed a case where this restriction has prevented the router from finding a path.

The other technique was suggested by Korn [Ko82]. It is implemented by making the distance from the cell to the goal part of the cost function as described above. The addition of this term means that the cost function reflects not only the cost of reaching the current cell from the origin (which is precisely known), but also the probable cost of reaching the goal from the current cell. When the maze routing algorithm is examining the wavefront list F, this will make it tend to choose the cell on the wavefront which is closest to the goal.

In practice, this technique provides only a modest speedup, because with the large number of obstructions in the channel, the closest cell to the goal is not necessarily on the path to the goal. If the path eventually found contains long jogs, the path will spend almost as much time moving away from the goal as moving towards it. Also, the adjustment to the cost based on the distance to the goal is relatively low compared to the other parts of the cost function. If the adjustment is too large relative to the rest of the cost, the router runs faster but produces inferior results.

The choice of high-level strategies can greatly affect the running time. Most of the successful routing is completed by the earlier stages, which run quickly because the left and right limits enclose a relatively small area. The later stages are slow-running, because they look at a large area, and they often fail to find a connection at all. If there is no connection possible, the maze router takes a long time before it gives up, usually considerably longer than when a path is found. This shows that adding additional stages to the high-level maze routing strategy will greatly slow the router down, while only occasionally improving the performance.

3.4. Experimental Results for Chameleon

Chameleon has been implemented in the C programming language, running under 4.2BSD UNIX. Table 3.2 gives the number of rows used for a suite of especially difficult benchmarks, using 2 to 6 layers with equal spacing. The channels labeled as **3a**, **3b**, and **3c** are from [Yo82]. **Diff** is the famous Deutsch's difficult example [De76]. **R1** through **R4** are random channels generated by Rivest's program [Ri82]; the channels have been generated so that they are of uniform density with many short nets making them difficult to route. **Chris1** is the largest channel of the CMOS implementation of SOAR (Smalltalk On A RISC) designed at Berkeley by Chris Marino. The channels marked **ex1** to **ex4** are difficult channels from a standard cell chip, and were provided by Dr. C.P. Hsu of Hughes Aircraft Company, Newport Beach. The examples **c1** to **c4**, **cycle.t**, and **bad** contain cycles in the vertical constraint graph.

The number of rows used by YACR using two layers is included for comparison. The entries in bold type are optimal. It can be seen that as the number of layers increases, it is easier to achieve optimal results. When there are four or more layers, the partitioning of the nets into groups reduces the number of vertical constraints and reduces the depth of the vertical constraint graph, making it easier to route a channel in density.

Table 3.3 compares the results of Chameleon with those of three other multi-layer channel routers. The label **C&L** refers to Chen and Liu's three-layer channel router

Example	Density	YACR	Number of Layers				
			2	3	4	5	6
3a	15	15	**15**	**8**	**8**	**5**	**4**
3b	17	18	*18*	*10*	**9**	**6**	**5**
3c	18	19	*19*	*10*	**9**	**6**	**5**
3cr	18	19	**18**	*10*	**9**	**6**	**5**
bad	4	8	*9*	*3*	*3*	**2**	**1**
c1	5	5	**5**	**3**	**3**	**2**	*3*
c2	4	9	*7*	*4*	**2**	**2**	*2*
c3	4	9	*7*	*4*	**2**	**2**	*2*
chris1	49	50	**49**	**25**	**25**	**17**	**13**
cycle.t	16	19	*17*	*9*	**8**	*6*	**5**
ddr	19	19	**19**	*11*	**10**	**7**	**5**
ex1	16	17	**16**	*9*	**8**	**6**	**4**
ex2	15	16	*16*	*9*	**8**	**5**	**4**
ex3	11	12	*12*	*7*	**6**	**4**	**4**
ex4	19	22	*22*	*11*	**10**	**7**	**6**
nasty5.5	7	8	*8*	**4**	**4**	**3**	**2**
nasty7.5	7	9	*8*	*5*	**4**	**3**	**2**
nasty7.7	9	11	*10*	**5**	**5**	**3**	**3**
nasty9.5	6	10	*8*	*4*	**3**	**2**	**2**
nasty9.6	6	9	*8*	*4*	**3**	**2**	**2**
nasty9.7	7	10	*8*	**4**	**4**	**3**	**2**
r1	20	22	*22*	*12*	*11*	**7**	*6*
r2	20	21	**20**	*11*	**10**	**7**	*6*
r3	16	18	**18**	*9*	**8**	*6*	**5**
r4	15	17	*17*	*9*	**8**	**6**	*5*

Table 3.2: Results with Uniform Pitch

[Ch84], the label **D&M** refers to a multi-layer channel router developed by Tony Ma and Srinivas Devadas [Ma85] of our group, and **M&B** refers to a router developed by Karti Mayaram, Jeff Burns, and Fabio Romeo [Bu85] of our group. This partitioning program used by Chameleon was derived from this router.

Table 3.4 compares the performance of YACR and Chameleon for a number of channels from standard cell layouts using two layers. Out of 47 channels (with sizes up to 640 columns), YACR was able to route 26 in density. The remaining ones are listed here. It can be seen that Chameleon was able to route almost half of these in density.

Table 3.5 gives the performance of Chameleon on the channels of a large standard cell circuit using 3 layers. The circuit is example **ind3** from chapter 4, and it has 19 channels and about 2400 cells. The data is especially relevant because current fabrication

Example	Density	3 Layers				4 Layers			5 Layers		
		Cham	C&L	D&M	M&B	CHAM	D&M	M&B	CHAM	D&M	M&B
3a	15	8	8	8	8	8	8	8	5	6	5
3b	17	10	10	9	9	9	9	9	6	7	6
3c	18	10	9	9	10	9	9	9	6	7	7
3cr	18	8		8	8	8	8	8	6	7	7
bad	4	3		11	4	3	3	3	2	3	3
chris1	49	25		25	25	25	25	25	17	17	18
diff	19	11	14	12	11	10	10	10	7	9	7
ex1	16	9		15	16	8	8	9	6	6	8
ex2	15	9		11	12	8	8	9	5	6	7
ex3	11	7		10	11	6	6	7	4	5	5
ex4	19	11		12	15	10	10	11	7	7	7
nasty5.5	7	4		5	5	4	4	4	3	3	3
nasty7.5	7	5		7	7	4	4	4	3	3	3
nasty7.7	9	5		7	7	5	5	5	3	3	3
nasty9.5	6	4		9	6	3	4	3	2	3	3
nasty9.6	6	4		9	7	3	4	4	2	3	2
nasty9.7	7	4		9	7	4	4	4	3	3	3
r1	20	12		11	11	11	11	10	7	8	8
r2	20	11		11	11	10	10	11	7	8	8
r3	16	9		9	9	8	9	9	6	8	6
r4	15	9		10	10	8	9	9	6	8	7
cycle.t	16	9		11	12	8	9	8	6	8	8
c1	5	3			5	3		4	2		3
c2	4	4			4	2		2	2		2
c3	4	4			4	2		2	2		2

Table 3.3: Comparision of Several Multi-Layer Channel Routers

technologies allow the use of three layers for routing. Three layers is also the most difficult number to use, because there will be more VCVs than with 2 layers, and the nets cannot be partitioned as when 4 or more layers are used. Note that Chameleon never had to use more than one row over the optimum number.

Table 3.6 compares execution times for Chameleon and other channel routers mentioned above. The execution times are for Deutsch's Difficult Example. These times do not include the time necessary to print the output, which can take a second or two, depending on the format. Chameleon requires less time for two and three layers because it is not not necessary to partition the nets. If the routing of the vertical segments without VCVs were done differently (as explained below), Chameleon would be about twice as fast for 2 or 3 layers, and about 33% faster for 4 or more layers.

Example	Density	Rows Needed	
		YACR	Chameleon
ind1.2	8	9	9
ind1.5	10	11	10
ind2.4	9	10	9
ind2.5	9	10	9
ind2.6	9	10	10
ind2.8	8	9	8
ind2.10	7	8	7
ind2.13	8	10	8
ind3.3	16	17	16
ind3.6	24	25	25
ind3.8	22	23	23
ind3.9	27	28	28
ind3.10	27	28	28
ind3.12	32	33	32
ind3.13	34	35	35
ind3.14	30	31	30
ind3.15	25	26	26
ind3.16	25	26	25
ind3.17	20	21	21
ind3.18	13	14	13
ind4.3	17	18	17

Table 3.4: Comparision of YACR and Chameleon on Standard Cell Channels

Table 3.7 gives an execution profiles of Chameleon when routing Deutsch's Difficult Example using 4 layers, and an industrial standard cell channel using 3 layers. Several points are of interest. First, a large portion of time is spent printing the results. This is because the relatively time-consuming *printf()* library routing is used, and the format is relatively elaborate. Second, a large amount of time is spent performing the trivial task of routing the columns without VCVs. This is because it was simpler to use the maze router for this task than to write a special routine to simply place unobstructed vertical segments in the channel. If less than four layers are used and the output is not printed, a majority of time may be spent doing this routing. Thirdly, only a small amount of time is spent routing the columns with VCVs. This can be deceiving. Other examples may have many more VCVs to be routed than there, which would increase the time required. Also, if the maze routing fails and a track must be added, the maze routing must be repeated. Other time-consuming steps, such as partitioning the nets and printing the output, need to be

Channel	Density	Rows	
		Optimal	Actually Used
1	10	5	5
2	24	12	13
3	16	8	9
4	16	8	9
5	17	9	9
6	24	12	13
7	16	13	13
8	22	11	12
9	27	14	14
10	27	14	14
11	32	16	16
12	32	16	17
13	34	16	17
14	30	15	16
15	25	13	14
16	25	13	13
17	20	10	11
18	13	7	7
19	12	6	6
Total		218	228

Table 3.5: Performance of Chameleon on Standard Cell Channels Using 3 Layers

No. of Layers	VAX 11-780 CPU Seconds			
	Chameleon	Yacr	Mayaram & Burns	Devadas & Ma
2	9.0	1.8	7.5	37.5
4	11.7		10.5	38.4
6	10.8		13.2	39.6

Table 3.6: Execution Times for Deutsch's Difficult Example

done only once.

When the example **ex1** was profiled using 2 layers, it was found that 32 percent of the CPU time was spent doing the maze routing, if the time spent routing the columns without VCVs and printing the output is excluded. This was because Chameleon had to add two rows before the routing could be completed.

Portion of Code	Per Cent of Run Time	
	diff 4 Layers	ind1.9 3 Layers
Read Input and Build Net Data Structure	1.6	5.0
Allocate and Initialize Channel Structure	2.2	2.5
Partition Nets	35.0	0.0
Build and Level Vertical Constraint Graph	1.0	3.5
Compute Cost Function	0.5	1.3
Choose Starting Column	0.3	0.9
Select and Assign Nets to Rows	3.5	12.2
Route Columns Without VCVs	18.1	23.4
Route Columns with VCVs	1.1	1.4
Verification of Completed Route	3.2	5.8
Output Routed Channel	33.5	44.0
Total	100.0	100.0

Table 3.7: Execution Profiles of Chameleon

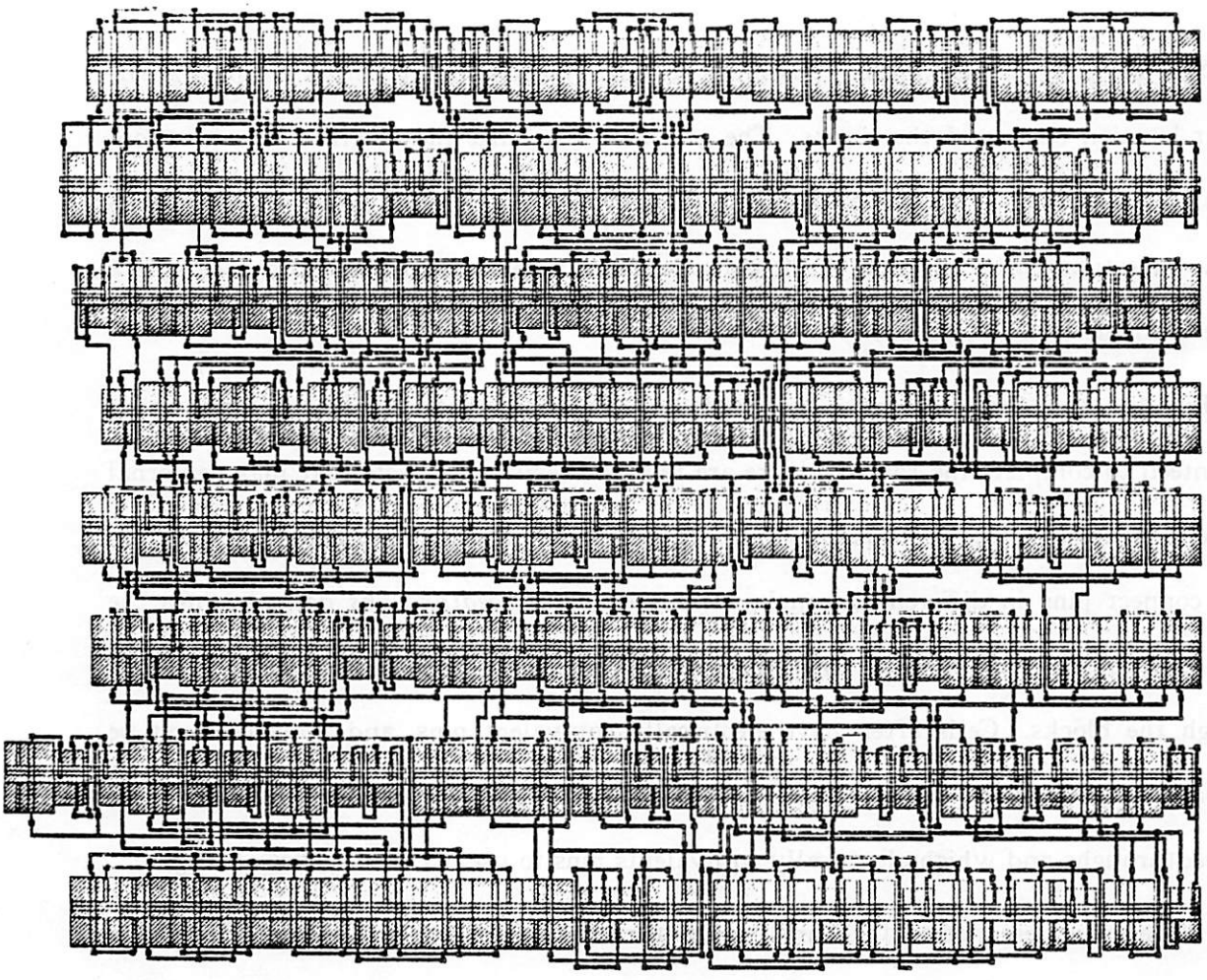
CHAPTER 4

The Standard Cell Place and Route System ThunderBird

4.1. Overview of Standard Cell Integrated Circuit Technology

The standard-cell design style [So81],[De76b] for integrated circuits has been popular since the advent of large-scale integration (LSI). It can be thought of as being a solution between gate-array and full custom design. A standard cell chip consists of a series of parallel rows, or *blocks*, of *standard cells*. The cells in a block are designed to be abutted in any order, and power and ground are usually fed to the cells through busses inside the cells established by this abutment. The spaces between the blocks form routing channels, which are routed by a channel router. The standard cells are usually pre-designed, and a *library* of them is maintained. The complexity of the circuitry in a standard cell is usually of the gate or register level. In this sense, standard cells are analogous to standard 74XX series TTL circuits, and it is not uncommon to have a library of standard cells that duplicates the functions of these circuits. There might be anywhere from a few to several thousand standard cells on one chip.

There are advantages to standard cell design that make it very adaptable to automation. First, since a library of cells is used, all the components of the circuit (except the wiring) have been design rule checked and have had their functionality verified and characterized. This makes the design and simulation of the circuit to be implemented much simpler and faster. Also, the layout of the circuit is much simpler than custom-cell layout. Assuming a quality channel router, it will always be possible to complete the routing. If there is insufficient space in any one channel to hold its wiring, the channel can be simply enlarged by moving the blocks further apart. In comparison, the custom-cell placement and routing problem is extremely complex. Gate-array design has a similar arrangement of blocks and channels, but the channels cannot be changed in size if there is



A Standard-Cell Chip

not enough room for the wiring. The density of standard-cell circuits is greater than gate-array circuits because the contents of each cell can be laid out as compactly as possible.

Since the channel size can be adjusted to always allow the routing to be performed, the problems of placement and routing can be done independently. The placement and routing of a standard-cell circuit can (ideally) be broken down into three steps.

First, the number of blocks of cells is decided upon, the length of the blocks is decided upon, and the channel width between each pair of blocks is estimated. The goal is to have the aspect ratio of the final chip fall within specifications. The total length of the blocks is equal to the total width of the cells, because the blocks are formed by abutting the cells.

Second, the *placement* of the cells must be done. Each standard cell is assigned a particular location on one of the blocks. The goal is to minimize the length of the interconnections between the cells, as well as the area needed by the wiring. The area of the wiring can be estimated by the sum of the densities of each channel, multiplied by the length of the channels (which is fixed).

Finally, the channels must be routed. Assuming that the channel router is guaranteed to complete the routing, there are few problems here. Nets that make external connections, for instance to pads, are routed through the ends of the channels. Nets that must connect pins in different channels are routed using *feedthroughs* designed into the cells, or through special feedthroughs inserted between cells that allow the net to pass through the blocks. Cells often have electrically equivalent pins, and a choice must be made about which one to use for a particular connection. The process of deciding where to put feedthroughs and which electrically equivalents pins to use is called *global routing*.

If the initial estimates of channel width turn out to be inaccurate, the final aspect ratio of the chip may no longer be within specifications and the configuration of the blocks will have to be changed, and the process repeated.

It is not necessary to implement an entire chip using standard-cell design. A portion of the circuit may be implemented as standard cells, while parts such as memories or PLAs may be done as custom cells. In this situation, the routed blocks of cells can be treated as a macro cell when doing the placement and routing of the chip as a whole. In this application, a macro cell made of standard cells has the advantage that its aspect ratio can be adjusted to optimize the placement and channel definition of the chip as a whole.

4.2. Overview of ThunderBird

4.2.1. Design Objectives

The goal of **ThunderBird** is to provide an automated standard-cell placement and routing system that will produce a ready-to-use layout from a circuit description. This is

a very complex task, for two different reasons. First, the problem is hard, both in the sense of being NP-complete and in the difficulty of devising suitable algorithms and heuristics. Second, in order to satisfy the goal of producing a ready-to-use layout, a large number of mundane tasks must be performed. In order to be confident that the final layout is electrically correct and satisfies design rules, the algorithms must be robust.

A tradeoff must be made between the generality and optimality of the program. If the program is being designed to work with a particular standard-cell technology and a particular design environment, the problem is more defined. The main purpose of ThunderBird is to be used here at U.C. Berkeley in our own design environment as part of an automated synthesis system. Unfortunately, the details of this system, such as the exact standard-cell technology and fabrication process are still undefined. It is also desirable to make ThunderBird as adaptable as possible to other design environments

In order to reach a compromise between these conflicting objectives, two principles were used in the implementation of ThunderBird. One is that any non-necessary part would not be implemented at this time. Some requirements of the system would be so technology-dependent that it would be pointless to implement them, because they would have to be rewritten for any particular application. On the other hand, technology differences that could be expressed as parameters, or that could be handled by a general algorithm, would be accommodated by ThunderBird. The other principle is that it should be easy to make the required modifications to adapt ThunderBird to any particular application, requiring a minimum amount of low-level recoding.

4.3. Description of ThunderBird

ThunderBird is implemented as a collection of four programs. The placement and global routing of ThunderBird is performed by the program **TimberWolf** described below. The channel routing is performed by the latest version of **YACR** described in section 2. The rest of ThunderBird consists of the netlist and cell data extraction program **Flounder**, and the TimberWolf post-processor and physical routing program **Termite**, which uses

YACR to perform the symbolic channel routing.

The sequence of operations of ThunderBird is as follows. Flounder is invoked to extract from the Squid CAD database system [Ke84],[Ke84b] the netlist and physical cell data needed by TimberWolf. It refers to a library (maintained by Squid) of the layouts of the standard cells used by the circuit to obtain the physical cell data. The result of this is a set of text files in the input format that TimberWolf expects.

TimberWolf is then executed with these files as input. When TimberWolf has finished (up to several hours later), Termite is invoked with the text files produced by TimberWolf as input. Termite creates a hierarchical layout description (in the Squid format) of the chip using the placement information contained in the input files. It refers to the library of standard cells mentioned above to obtain the layouts of the cells. It generates the descriptions of the channels to be routed, and repeatedly executes YACR to obtain the routed channels in symbolic form. For each channel, it generates the physical wiring according to user-specifiable design rules, and places the routing in the layout. The blocks of cells are compacted or expanded as necessary so that the channels will always be as narrow as possible. The final result is a physical layout description of the entire circuit.

4.3.1. Notable Features of ThunderBird

ThunderBird has several advanced features that increase its usefulness and improve the quality of the results. We have tried to add features that would extend its generality, instead of being useful only for a particular technology. ThunderBird exploits several features of TimberWolf. TimberWolf is able to take advantage of electrically equivalent pins in cells, as well as built-in feedthroughs. It will also allow the user to specify pads and a limited number of macro blocks in the circuit. The global router does extensive optimizations to reduce the total density of the channels. Because of the irregular channel feature added to YACR, ThunderBird will correctly handle standard cells of varying height, and will use the resulting indentations in the channels to hold nets. It is also possible for the user to specify the design rules for the wiring. The user needs to specify the

width and self-separation of the two layers used for routing, along with the height and with of a contact between them. ThunderBird will use this data to space the wiring in the channels as tightly as possible without causing spacing violations.

A notable feature is that the pins may be at arbitrary locations on the cell edges. ThunderBird is gridless, and it will perform the translation from the symbolic grid used by YACR to the actual pin locations of the cells.

ThunderBird will also extract the physical data from the standard cells (their sizes and pin locations) directly from the layout data. This reduces the amount of input data required, and reduces chances of data inconsistency problems.

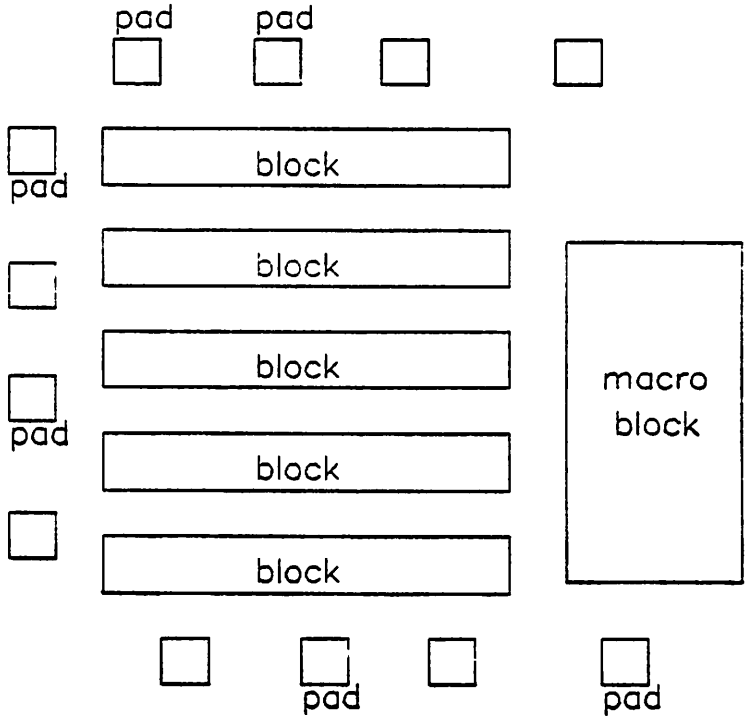
4.3.2. Limitations of ThunderBird

In its current implementation, ThunderBird does not perform every step needed to produce a complete layout. The steps omitted are generally quite technology-dependent and/or easy to perform manually. When ThunderBird is optimized for a particular design style, the current steps that must be done manually can be added with relative ease to the code.

One limitation is that ThunderBird does not produce the initial floorplan of the chip. It will be necessary for the user to decide how many blocks are necessary to hold the cells, as well as the relative positions of the pads and macro blocks. It is also necessary for the user to come up with an estimate of the average spacing necessary between blocks, based on the expected channel densities. This is only an estimate to optimize the operation of TimberWolf; ThunderBird will always pack the blocks as tightly together as possible.

Another limitation is that only parallel blocks of cells may be used. This is a limitation of TimberWolf; having both horizontal and vertical blocks would greatly complicate the global routing. Also, all macro cells must be completely outside of the blocks of cells. In other words, the blocks of cells must form a single rectangle with no other cells within its bounding box. If macro blocks were allowed to be within the block area, many subtle

and nasty problems would spring up. For example, the fact that ThunderBird packs the blocks together would greatly disturb the relative positions of the cell blocks and the macro blocks, as well as the channels between them.



A Legal ThunderBird Floorplan

Not all of the routing is placed. Only the wiring connecting the standard cells in the channels is placed, not the connections from the standard cells to the pads or macro blocks. The wires to the pads and macro blocks are instead brought out to the edge of the rectangle bounding the standard cell blocks to a point as close as possible to their final destination, and are labeled with the signal they represent. This is done for two reasons. First, the area just outside the standard-cell blocks will almost certainly have to have power, ground, and possibly clock wiring placed in it. This type of wiring almost always has to be routed in a special fashion, and must be placed before any other wiring. It is pointless for ThunderBird to route the signal nets in this area, since it has no way to keep their wiring out of the way of the power and ground wiring. Also, the blocks will be packed, so their relative positions with respect to the pads and macro cells will change.

Since ThunderBird does not know how the pads and macro cells may be legally moved about, there is no point in placing the wiring to them. If the pads or macro blocks were later to be moved, all this wiring would have to be ripped up anyway.

As inferred to above, ThunderBird does not attempt to do power and ground routing. This is quite technology-dependent, and the addition of sufficiently general-purpose power and ground router would greatly increase the complexity of the program. For any particular technology, especially given the constraints on block placement mentioned above, adding a technology-specific power and ground router would be much simpler.

4.3.3. ThunderBird and Squid

Squid is an electronic circuit CAD database management system [Ke84],[Ke84b]. It provides a data model and access routines for all types of IC CAD data. It is capable of handling physical layouts, connectivity information, or other data in a uniform way. From the point of view of the programmer, Squid is a library of C routines that allow access to the database. Although the data is stored in files, the programmer does not read or write them directly, but instead makes calls to Squid to access the data. The types of Squid objects that ThunderBird uses are conceptual objects such as *nets* and *terminals*, as well as physical objects like rectangles of metal or poly. Relationships can be established among these objects. For example, a terminal can be associated with a particular net, and a square of metal can implement a particular terminal. Squid is hierarchical; a *cell* can contain *instances* of other cells. A cell may have several versions, or *views*. For example, there may be a *physical* view of a cell that contains its layout, a *symbolic* view that has a symbolic description of its layout, or a *body* view that has a graphic representation of the cell for use in schematic diagrams.

The use of Squid to manage ThunderBird's data reduces the generality of ThunderBird. Squid was used because we wanted to integrate ThunderBird into the design environment here at Berkeley. Also, ThunderBird had to run in a design environment of some sort to produce test cases.

It would be relatively easy to remove the dependency on Squid from ThunderBird. ThunderBird interacts with Squid in two places. The first is when it reads the circuit connectivity in Flounder, and the other is when Termite writes out the layout data.

Flounder is quite dependent on the Squid methodology of representing the connectivity. If Squid were not used, Flounder would have to be discarded. Flounder is essentially a preprocessor for TimberWolf and operates independently of the rest of ThunderBird, so this would not be a problem. In fact, the circuit data for our industrial test cases was not in Squid format. To run them, it was necessary to generate a Squid library of dummy standard cells. Termite was then able to use the TimberWolf output files and the dummy cell library to produce the finished layout.

Termite is less dependent on Squid than Flounder. It would be easy to modify Termite to write the physical layout data in some layout description language other than Squid, such as CIF or EDIF. This is because Termite only calls Squid to place instances of cells and pieces of geometry into the final layout. The routines to do this are quite well modularized and could be quickly replaced with routines to output textual data in some layout description language. There are also some calls to Squid routines to initialize the Squid package, etc., but these would have no counterpart outside of Squid and could simply be removed.

4.4. Overview of TimberWolf

TimberWolf is a suite of programs, two of which perform placement and global routing of standard-cell integrated circuits, developed by Carl Sechen [Se84],[Se85]. It forms the core of ThunderBird.

TimberWolf has the ability to take advantage of electrically equivalent pins in cells as well as built-in feedthroughs. It can also allow the inclusion of pads, as well as a limited number of macro blocks in the placement. The global router does many optimizations to reduce the wire length and channel densities.

Several aspects of TimberWolf have made it easier to build a complete interface around it. First, TimberWolf is extremely technology-independent. It treats standard cells in an abstract manner, essentially as boxes with wires to be connected to them. It has no built-in assumptions about the technological details of the standard cells being used. It is gridless; pins and cell borders may be arbitrarily located. It deals with technology-specific concepts such as built in feedthroughs and pins in strange locations on the cells in a manner that makes it adaptable to most standard-cell technologies. It does not need to know anything about the material used for wiring, or its design rules.

The input to TimberWolf is contained in several text files. One file (the **block** file) contains a description of blocks of cells and their relative locations. Another file (the **cell** file) contains a description of each cell, macro block and pad in the circuit. The information includes the height and width of each cell, and the x and y coordinates of each pin, as well as the net that each pin is connected to. Electrically equivalent pins can also be specified, and TimberWolf will choose the best one to use for the actual wiring. It is also possible to specify uncommitted feedthroughs in the cells. The global routing stage of TimberWolf will decide which nets will use these feedthroughs. If there are no feedthroughs built into the cells, or if there is not one in a desired location, the global router will leave room between two cells of a block to insert a feedthrough. Other input files describe net weighting information and allow the user to specify parameters for the operation of TimberWolf.

TimberWolf performs placement by using *simulated annealing* [Se84], [Ro85],[Ki83]. Although the details of the algorithm are contained within the program, and will not be discussed in depth, certain implications of the algorithm affect the operation of Thunder-Bird. Simulated annealing optimizes the placement of the cells (expressed as a cost function based on the total wire length) by performing a large number of interchanges and moves of cells. The constraints within which cells may be interchanged or moved, and the conditions under which a move is accepted or rejected, are a critical part of the algorithm.

For a relatively large standard-cell chip (1000 cells), approximately 35 million moves might be attempted. This would require about 8 hours of VAX CPU time. About 1000 bytes of memory per cell are required, so the memory requirements are relatively large.

After TimberWolf has optimized the placement of the cells, the global router is used. It decides where to put feedthroughs (or use built-in feedthroughs) to connect cells in blocks that do not share a channel. If a net can be connected to any one of several electrically equivalent pins, a particular pin is decided upon. The above steps are designed to minimize wire length.

A second phase of the global router attempts to shift the blocks and swap cells in order to minimize the density of the channels between the blocks. If due to the presence of electrically equivalent pins on the top and bottom of cells, a portion of a net can be placed either above or below a block, the global router makes the choice that minimizes channel density. Experience has shown that the total channel density can often be reduced by up to 20% by these techniques.

The output of TimberWolf is contained in three text files. One contains the final coordinates of the blocks, and by implication the coordinates of the channels between them. Another file contains the final coordinates of each cell, pad, and macro block. If the cells have been mirrored or rotated, this fact is noted. The third file contains the results of the global router. Because of the presence of feedthroughs (either built-in or inserted by TimberWolf) and electrically equivalent pins, all the pins of a particular net are not necessarily connected directly together. Instead, *groups* of pins are connected together. A net may consist of several groups, which are made electrically equivalent by sharing two electrically equivalent pins of a cell, or the top and bottom pins of a feedthrough. The output file of the global router consists of a list of each pin that has a connection, along with the cell it belongs to, the channel it is in (if any), whether it is at the top or bottom of the channel, the name of its net, and its group number. When the channel routing is done, all pins with the same group number are to be connected together, and the original

net name is ignored. Information such as the cell name and channel number is included so that it is not necessary for the routers to refer back to the TimberWolf input files to extract it.

4.5. Flounder

Flounder is a program that generates the input data for TimberWolf from the Squid database. The circuit-specific data that TimberWolf requires is conceptually simple: a list of all the cells that make up the circuit, with their size, the positions of their pins, and the net that each pin is connected to. The problem is that this information is 'flat' and non-structured. Different parts of the information are unrelated to one another and would have to derived from different sources. For example, the location of the terminals of a standard cell is totally unrelated to the connectivity description of a circuit using the cell. The two types of data would very likely be managed by different persons, and possibly different software packages.

The data required by TimberWolf that Flounder must assemble can be divided into two types: information that describes individual cells, and information that describes the relationship between cells. The information describing an individual cell consists of the dimensions of its bounding box, and the location of each of its pins. The inter-cell information consists of a description of the connectivity of the circuit. This is described by naming the net that each pin (or set of electrically equivalent pins) is connected to.

The data describing one cell consists of dimensions and locations that are determined by the physical layout of the cell. The nature of the Squid database allows Flounder to extract this data directly from the physical layout of the cell. The advantage of this is that there will never be a consistency problem. If any design changes are made to a cell that affect its size or pin locations, Flounder will always find the up-to-date data.

The dimensions of the cell are derived from the bounding box of the cell's physical layout. If this is not appropriate, the designer can place a rectangle on a special symbolic

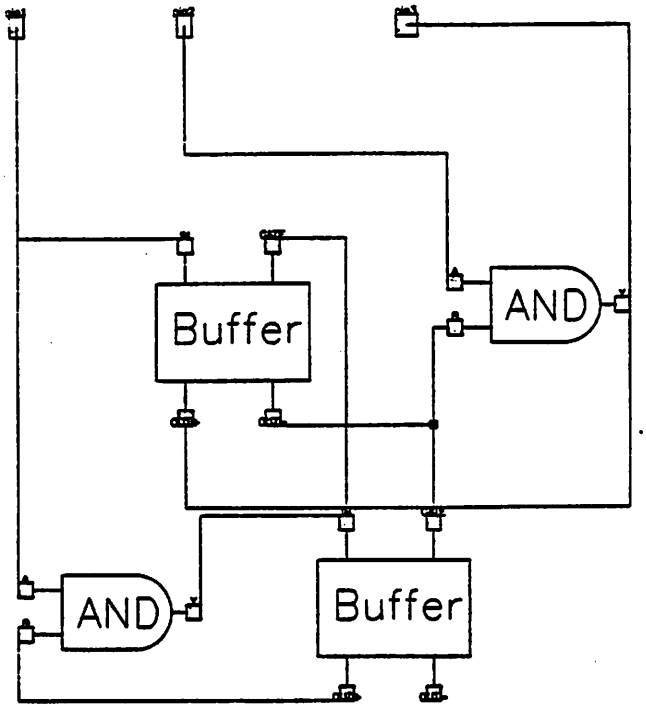
layer to define the cell border that Flounder should use.

Squid simplifies the determination of the locations of the terminals. In Squid, a *terminal* is a conceptual object. It is given physical meaning by creating pieces of material that *implement* the terminal. Squid provides a routine to quickly generate all the geometries that implement a particular terminal. If exactly one geometry exists to implement a given terminal, its center determines the coordinate of the terminal. If there is more than one piece of geometry, each piece represents an electrically equivalent terminal.

Extracting the connectivity information is more complex. This is because TimberWolf treats the data much differently than a circuit designer would. To TimberWolf, each cell in the circuit is a unique entity. Also, there is no notion of hierarchy in the description of the circuit. If a hierarchical description of a circuit is created, each cell and net no longer has a unique representation. For example, if there are several instances of a cell in a circuit, each subcell of the master of the cell is only represented once. To create a unique object for each cell in the circuit, the description of the circuit must be *flattened*, and the hierarchy expanded.

Since the most common graphical representation of connectivity is via a schematic diagram, and the Hawk/Squid environment already contains the facilities necessary to create schematic diagrams, it is assumed that the connectivity would be derived from this source. Since Flounder is only interested in the abstract information (nets and terminals) instead of the physical information (positions of lines and figures on the diagram), there is little in Flounder that requires an actual schematic to be used. In fact, any Squid view that expresses the connectivity of the circuit in terms of nets, terminals, and instances can be used by Flounder to extract the information necessary to run TimberWolf.

There is one peculiarity of the schematic diagrams in Squid that Flounder adapts to. A schematic diagram of a circuit is by convention a *schematic* view. However, the subcircuits of the schematic are represented by *body* views. This is because the viewer of the schematic wishes to see a box or some symbol (like a logic gate) to represent the subcircuit.



A Hawk/Squid Schematic Diagram

The body view is simply a graphic icon that represents the subcircuit. The actual connectivity of the subcircuit is contained in a schematic view of the subcircuit. To traverse the hierarchy of cells or subcircuits in a Squid schematic view, the following algorithm is used. Examine every instance in the current schematic view. Each of these instances is presumably a schematic view. For each instance, see if a schematic view exists. If so, recursively start traversing the schematic view of the instance. If not, the body view must represent a primitive cell of the circuit. Find a view containing the physical layout information (a *physical* view or *tbird* view), and use it to create a new leaf cell to be given to TimberWolf.

4.5.1. Implementation of Flounder

Flounder recursively traverses a hierarchy of Squid views, while merging the nets of the circuit represented by the view. When a leaf cell of the hierarchy is encountered, the physical data describing the cell (bounding box and pin locations) is extracted. When the entire hierarchy has been traversed, the connectivity of the circuit is known, and the list

of leaf cells encountered, along with the nets that each of their pins is connected to, is dumped out in the format expected by TimberWolf. Squid maintains a stack of views; this complements the recursive nature of the algorithm. To simplify debugging and to reduce the number of (relatively costly) calls to Squid access routines, various lists and trees are also used to store the partial result. Most significantly, a list of all nets discovered so far, each with a list of all its terminals, is maintained.

It is assumed that the reader is fairly familiar with the Squid database concepts. It would be wasteful to give a complete description here. Instead, some especially subtle or confusing points are mentioned as necessary.

The algorithm runs as follows. The schematic view of the root cell of the circuit hierarchy is placed on Squid's stack of views. Each *formal* terminal of the root cell (the terminals that are its interface to a higher level of hierarchy, as opposed to the terminals of the instances it contains) is assigned to a net, which is added to the list of nets. The recursive routine is now called.

The recursive routine works with the schematic view that is currently on the top of Squid's stack of views. First, it generates all the formal and actual terminals (*actual* terminals are the terminals of the sub-cells of a view) of the current view. If the terminal is a formal terminal, it is added to the net in the net list that contains the corresponding actual terminal in the view's parent. If the terminal is an actual terminal, a new net is created for it in the net list.

The nets in the list of nets must now be merged. All the nets in the current view are generated by Squid. For each net, all of its terminals are generated. The net in Flounder's net list that each terminal is connected to is found, and all of these nets are merged into one.

Now, all of the instances in the current view are generated. Each of these views is presumably a *body* view. If a corresponding *schematic* view exists, it is pushed onto Squid's view stack, and the recursive routine is called to continue the processing. If a

schematic view does not exist, it is assumed that we have found a leaf cell of the hierarchy, and there must be a standard cell corresponding to the body view. Therefore, a *tbird* view corresponding to the current view is searched for. An entry is placed in Flounder's list of cells, given a unique ID, and the bounding box and pin locations are extracted from the layout data contained in the tbird view.. If a tbird view does not exist, a fatal error is generated.

Once the recursive algorithm has run its course, all the necessary data is stored in Flounder's data structures. This data is now dumped out in the format that TimberWolf expects.

4.6. Termite

Termite is the program that generates the actual physical layout of the circuit, based on the placement and global routing data output by TimberWolf and the contents of the cell library.

The operation of Termite is conceptually simple. Each block of standard cells, and each pad and macro block, is placed in the correct position on the chip. For each channel, the nets with pins in the channel are collected, and the symbolic routing problem is written to a temporary file. YACR is invoked, the symbolic routing it generates is read in, and pieces of metal and/or poly and contacts are placed on the chip to implement the routing.

Several problems make the actual implementation more complicated than it may seem. First of all, Termite does not expect to find the terminals on fixed grid locations. This means that a virtual grid must be established over the channel, and the pins must be assigned to columns on the virtual grid. It will generally not be possible for the x coordinate of each pin to lie exactly on a grid line. This means that jogs will have to be added to connect pins to their vertical wiring segments, which lie exactly over the grid lines. It is also necessary to ensure that in every case there are no violations of design rules. Because the design rules can be specified by the user, the results must be correct for any reasonable

set of design rules. There are many places in the program where extra processing has to be done to make sure there is no possibility of design rule violations. Also, Termite compacts the blocks of cells as it proceeds. This is done to make the channels exactly as wide as necessary to hold the finished routing. This means that Termite's internal database must be updated after every channel is routed.

4.6.1. Input to Termite

The input to Termite is the files generated by TimberWolf and the standard-cell library. Termite pays no attention to the contents of the standard-cell layouts. They must be present only so that they can be placed in the final layout. The files generated by TimberWolf were described above. They give the final location and orientation of each block and cell, and the net group that each pin is connected to. One file (the **.pl1** file) gives the coordinates and orientation of each block of cells. This information is used to determine the extent of each channel, which is simple because only horizontal blocks are allowed. Another file (the **.pl2** file) contains the location and orientation of each standard cell, pad, and macro block, as well as the block that each standard cell belongs to. This is used by Termite to place each cell in the correct place on the final layout. The third file (the **.pin** file) gives the net group that each pin of the circuit is connected to. Termite uses this to assemble the input for the channel router YACR. The final x and y coordinates of each pin are included in this file, so that Termite does not have to re-extract their locations from the standard cell layouts.

4.6.2. Output of Termite

Termite's output consists of a single hierarchical Squid view. The view contains all the geometry that makes up the cells and the routing. The hierarchy is decomposed as follows. The top-level view contains an instance for each block, pad, macro block, and channel. If there are N blocks, there will be $N+1$ channels, one between each block, one above the top block, and one below the bottom block. The block and channel instances are

stacked in alternating fashion, and are packed together.

Each pad or macro block instance is taken directly from the cell library, and is unchanged by Termite. Each instance that represents a channel contains the rectangles of metal and poly (the actual layer names are specified by the user) that implement the wiring, along with a number of instances of a contact cell. Each instance that represents a block contains only sub-instances of the cells contained within the block.

This simple arrangement of the blocks and wiring is possible because TimberWolf's global router has decomposed the routing into independent problems, one for each channel. If a net connects two pins on widely separated blocks, TimberWolf inserts feedthroughs in all the intermediate blocks (or uses built-in feedthroughs), so that the net is divided into a series of sub-nets, or *groups*, each composed only of pins in the same channel. This allows termite to deal with the routing of each channel independently. As mentioned above, nets that have connections outside the channels (such as nets connected to pads) are only partially routed by Termite. In this case, the portion of the net within a channel is routed, and a terminal is created on one end of the channel.

4.6.3. Operation of Termite

The high-level operation of Termite is straightforward. First the input data files are read. Then blocks of cells and routed channels are alternately created and placed in the chip's Squid view. Finally, the pads and macro blocks are placed. A more detailed description of the operation follows, with the next sections devoted to describing the channel routing and channel definition algorithms, the most complex parts of Termite.

Termite first reads the user-specifiable parameters. These specify the design rules that termite is to follow when creating the routing. As mentioned above, they include the width and self-separation of each layer, and the width and height of the contacts. This data is stored in the **.cadrc** file. This is essentially a database that provides technology and user-specific information for many CAD tools developed at Berkeley. A pre-existing

set of access routines are used to extract the data needed by Termite from the file.

The files produced by TimberWolf are then read in. Each file consists of a series of records of identical format, one per line, so the parsing of the data is trivial. The data contained in them is stored essentially unchanged in memory. This is because the data in the files is not sorted, so they must be repeatedly scanned for each block or channel that is constructed. Profiling of Termite has shown that the time saved by sorting the data after it is read in, is insignificant. The y coordinates of every cell, block, and pin are then adjusted to place bottom of the lowest block on the x axis. This gives a fixed reference point for use when packing the blocks.

The Squid database package is then started, and a Squid view is created to hold the layout. Now each block and channel is created. When the blocks and channels have been created, and the pads and macro blocks are placed, all that remains is to shut down Squid, causing all the layout data to be written out to the file system.

To create a block, Squid is called to create a view to hold the block. The cell list is then scanned, and every cell belonging to the current block is placed in this view. The location and orientation provided by TimberWolf for each cell is used to calculate the coordinate transformation that will place it in the correct position.

There are two things that complicate the creation of the blocks. One is that the feedthroughs added by TimberWolf appear in the .pl1 file (which was read in to form the cell list) as a cell with the symbolic name "twfeed". Since different technologies will need different feedthrough cells, the user may specify (in the .cadrc file) the actual name of the cell to be used for feedthroughs. The other is that the position of a cell's bounding box in the cell's coordinate frame must be known. While TimberWolf requires that the bounding box be centered around the origin, it may be at any location in the cell's actual layout. Therefore, the cell's layout must be examined to find the actual center of its bounding box. This is the only time that Squid data is read by Termite.

4.6.4. Termite's Channel Routing.

The portion of Termite that performs the channel routing is about two-thirds of the program. It is based on a program written by Deirdre Ryan [Ry85], establishing an interactive interface between YACR and the Hawk graphics editor. Integrating it with Termite, and adding the extra features required by Termite required an almost total rewrite. The most significant modifications were the addition of support for irregular channels and improvements in the algorithm for establishing the virtual grid for the routing. Since the original version was intended for interactive use in a layout editor, tradeoffs were made that improved the routing density, but failed to guarantee that there would be no design rule violations. Since this is unacceptable in an automated system such as ThunderBird, many additions and changes were required to guarantee that design rules would always be satisfies using unnecessary space for the wiring. Any further improvements in the compactness of the wiring would probably require a general-purpose compaction program, which would be considerably more complex than the rest of ThunderBird.

The steps to route one channel are as follows. First, every pin on the channel of interest is found in the list of pins (built from the contents of the .pin file), and used to construct the top and bottom pin lists of the channel. Although these are ordered, the exact grid locations of each pin have not yet been determined. Accordingly, there are no null terminals (representing unused grid positions) in the lists. It must then be determined which nets with pins in the channel also exit the channel, and end pins are created for these nets. The left and right pin lists are created from this information.

The vertical lines of a virtual grid are now established. This means that the actual x coordinate of every column of the routing is established. The width of each column is not the same, and the positions of the columns are adjusted to align as closely as possible to the actual locations of the pins. After this is done, new top and bottom lists are generated, with the unused pin positions represented by null terminals. The following section contains a more detailed description of the grid definition process.

Since ThunderBird is designed to handle irregular channels, the top and bottom offset lists needed by YACR to define the shape of the sides of the channel must be calculated. The offset of each column containing a pin is determined by the y coordinate of the pin, while the offsets of columns not containing a pin are found by finding the location of the top or bottom of the cell that abuts the channel at that point. The cell list is scanned to obtain this information. Care must be taken to ensure that there will be no design rule violations.

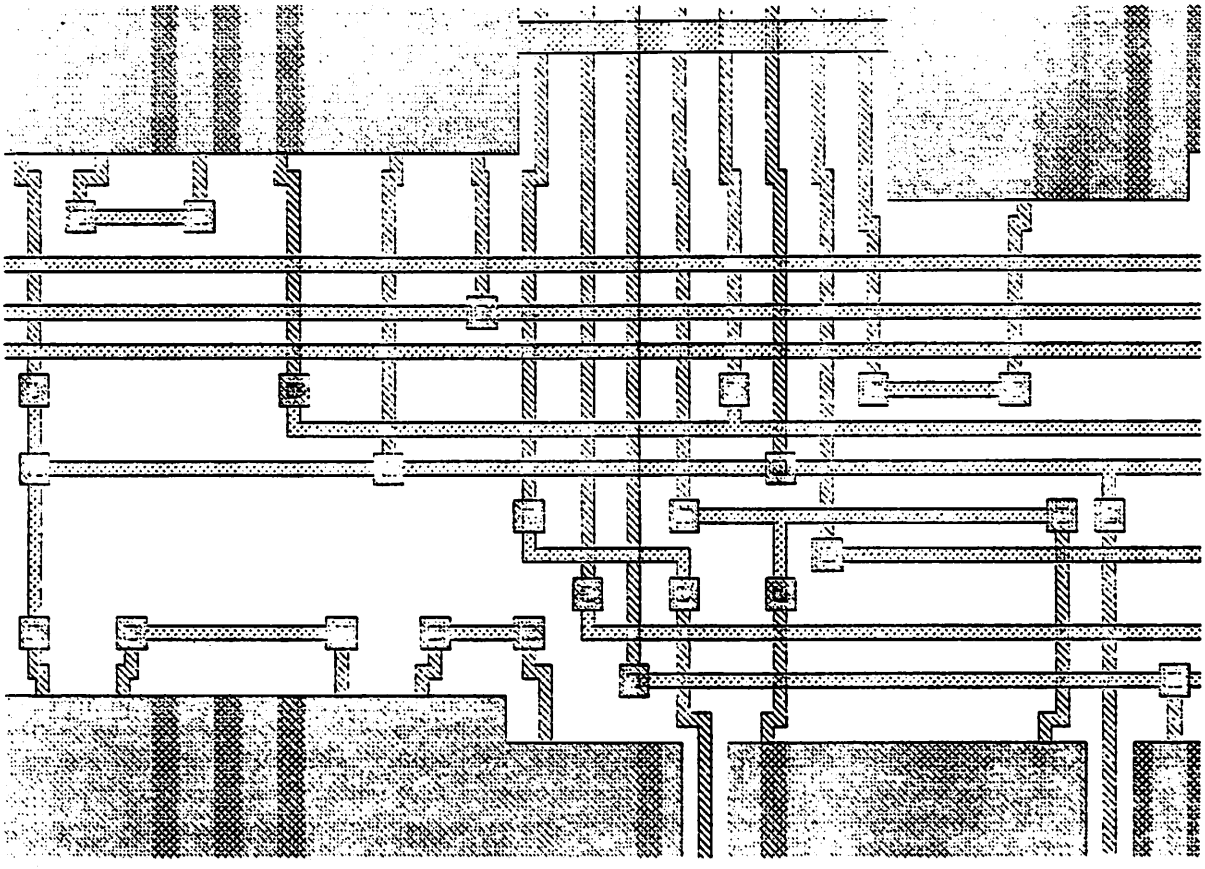
The input file for YACR is written, and a YACR process is spawned off to perform the symbolic routing. When YACR is finished, its output file is read in.

It is now necessary to define locations of the horizontal lines of the virtual grid, and thus the actual position of each row in the channel. This is not done until after the channel is routed because it is not known until then how many rows the channel will have. Also, the location of the horizontal and vertical segments and contacts are now known, and this information can be used to pack the rows of the channel as tightly as possible without causing any design rule violations. Actually, the row locations of the rows lying within indentations of the channel (if it is irregular) were already defined before the channel was routed, but no optimization could be done on them.

The data output by YACR is now searched for horizontal and vertical segments, and Squid is called to place each one, along with the contacts between them, into the Squid view of the channel. Pieces of material are also added to connect the actual location of each pin to its grid location. These are required because the y coordinate of a pin will not necessarily be on a horizontal grid line. If the pin is not directly over its vertical grid line, these pieces create the horizontal jog from the pin to its column. Once this is done, memory allocated for the various lists and arrays is freed, and the routing is finished.

4.6.5. Grid Definition in Termite

YACR is a grid-based channel router. This means that it treats the two-dimensional



Portion of Routed Channel

channel as a two-dimensional array of *cells* of equal height and width. Each pin is assumed to lie at the centerline of some vertical column of cells. Thus the x coordinate of a pin is represented by the *column number* of the cell that it is adjacent to. The horizontal and vertical segments of the wiring are assumed to run along the centerlines of the rows or columns of cells. This way, the vertical segments will exactly intersect the pin that they connect to.

To translate from the actual physical coordinates of the pins and channel to the abstract grid coordinates, the actual width and location of each column and row of cells must be defined. The rows and columns of cells can be dealt with independently. As described in the previous section, the locations of the columns of cells must be established before the channel can be routed. Since the width of the channel is fixed, and the height is

adjustable, the definition of the columns is more difficult. The rest of this section discusses the definition of the columns of the grid.

The problem of defining the locations of the columns can be more precisely stated. A column of cells can be defined by its width and the x coordinate of its left edge. There is some minimum value of the width W_{min}, which is established by the maximum pitch of the layers being routed in the column (including allowance for adjacent contacts). The columns may not overlap, but there may be empty space between them. Therefore, we must create an ordered list L of the x coordinates of the left edges of each column, and an ordered list W of column widths, such that for every column number j, $L_j < L_{j+1}$, $W_j \geq W_{min}$ and $L_j + W_j \leq L_{j+1}$. Each column may have at most one top and one bottom pin located within its borders. Ideally, each pin would be centered on a column; i.e., for any pin's x coordinate P_j, $P_j = L_j + W_j / 2$.

If every top and bottom pin on the channel had an x coordinate that was an exact multiple of W_{min}, the establishment of L and W would be easy. However, ThunderBird assumes that pins may have arbitrary x coordinates. It may often be possible to set up the grid so that pins are centered in their cells (especially if the distance between pins is large compared to W_{min}), but generally it will be necessary to add jogs of material to connect the vertical segment at the center of the cell (whose x coordinate is $L_j + W_j / 2$) to its pin's actual location (whose x coordinate is P_j).

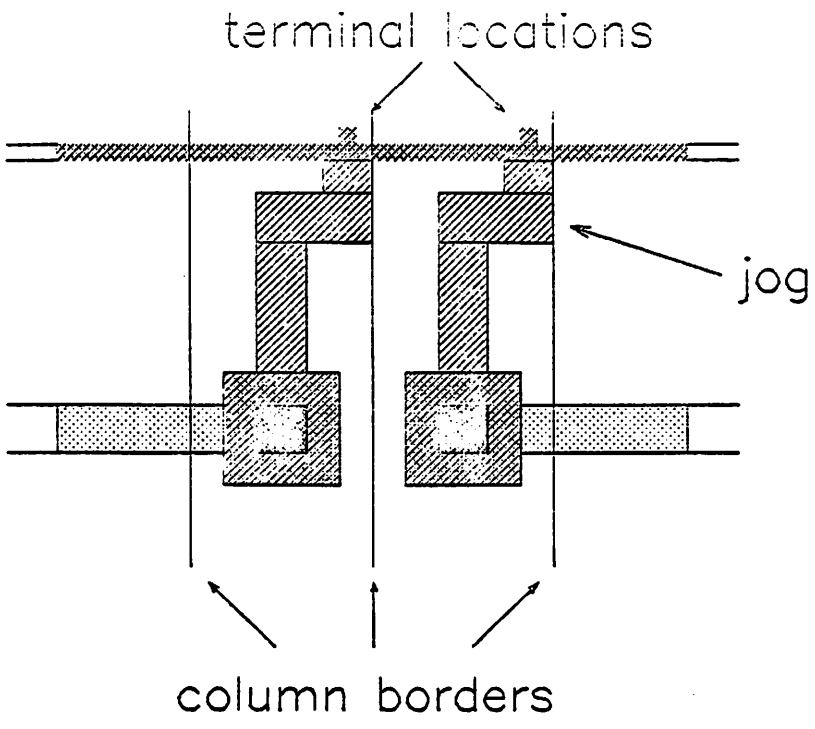
It is necessary to make sure that the material used to make these jogs does not create a design rule violation by getting too close to the vertical segment in an adjacent column. The constraints that this implies can be stated (for the left and right of the column, respectively) as:

$$L_{j-1} + W_{j-1} / 2 + S \leq P_j$$

and

$$P_j + S \leq L_{j+1} + W_{j+1} / 2$$

where S is the pitch spacing (width plus self-separation) of the material being used



Grid Locations and Jogs

($W_{min} \geq S$), and P_j refers to the x coordinate of either the top or bottom pin of the column.

It is the goal of the grid definition routine to come up with the list of cell locations L and widths W that satisfy the above criteria. It may not always be possible to succeed. If several adjacent pins have a separation less than S, it will obviously be impossible. If $W_{min} = S$, which will happen if it is not necessary to allow for contact-contact separation, it will be impossible to have any jogs, and the top and bottom pins in any column must have the same x coordinate. Even if W_{min} is larger than S, an unfavorable arrangement of tightly packed top and bottom pins may not be able to have a grid fitted to it.

The algorithm used by Termite to establish the grid sweeps across the channel from left to right, establishing the location and width of each successive column. The location of the left edge of the first cell L_0 is set to be the coordinate of the leftmost pin of the channel, minus half of the minimum cell width W_{min}. Now assuming that L_{j-1} and W_{j-1} have been established, the following algorithm establishes the successive cell locations.

L_j is set to $L_{j-1} + W_{j-1}$. If the next unassigned pin is to the left of L_j, the pin cannot be assigned a grid location. It is deleted from the list of pins, and we continue without it.

W_j is set to W_{min}. If the next pin is still to the right of the column (i.e. $L_j + W_j < P_n$), we have created an empty column. This is desirable because it gives YACR's maze routers room to work in.

If the next pin is within the column, we extend the right border of the column by increasing W_j until the pin is far enough inside the column to satisfy the jog spacing clearance constraints mentioned above.

We now look at the pin across the channel from one we just considered. If it is within the column, we may have to extend the right border of the column even further until the pin is far enough inside the column.

Now that the left border L_j and the right border $L_j + W_j$ have been established, we try to improve their location. It would be nice to have the pins centered in the column (i.e., $P_j + L_j + W_j / 2$), because jogs of material would not be required to connect pins to their vertical segments. If there are both a top and bottom pin in the column, their x coordinates must be identical to do this. Since the columns to the left are already fixed, we can only shift the current column to the right. If the distance between the right border of the column and the next pin in the channel is sufficiently large, we try to shift the column to the right to center its pin(s).

We now continue with the next column.

The above algorithm does not guarantee to find a legal placement of the columns if one exists. To make the algorithm more robust, it would be useful to abandon the one-pass approach it uses, and rearrange several adjacent columns at once. One feature of TimberWolf that helps it succeed is that TimberWolf will try a horizontal shift of the blocks to align the pins at the top of the channel with those on the bottom. For some industrial examples we tried, it would not have been possible to create any grid if TimberWolf had

not done this. This was because their design rules specified that contacts were no wider than the material being contacted. This allowed W_{min} to be the same as S, which prevented any jogs from being used (as shown above). Also, W_{min} could not be explicitly set to be larger than S, because there were large numbers of adjacent pins with spacing S.

4.7. Experimental Results for ThunderBird

The performance of ThunderBird depends on the quality of the placement done by TimberWolf, and the routing done by YACR. Since these steps are done independently, the results of the placement and routing can be examined separately. We have run ThunderBird on 5 industrial test cases, and 2 artificially generated test cases. The number of cells ranged from 200 to about 2300. Table 4.1 gives the characteristics of the industrial circuits.

Circuit	Cells	Blocks	Total Terminals	Total Tracks
ind1	286	10	1648	95
ind2	469	12	1891	115
ind3	2357	18	11356	436
ind4	207	3	1112	53
ind5	800	14	4683	298

Table 4.1: Industrial Standard Cell Circuits

Table 4.2 compares the results obtained by ThunderBird for the examples that had already been placed by the originizations that supplied them. The total wire length (in arbitrary units), the sum of the channel densities, and the number of feedthroughs (explicit or builtin) required are compared for the previous placements and ThunderBird's placements.

Circuit	Cells	Total Wire Length			Total Channel Density			No. of Vias		
		Prev	TBird	Red.	Prev	TBird	Red.	Prev	TBird	Red.
ind2	409	648,936	542,850	16%	141	109	23%	93	76	18%
ind3	2357	4,669,356	2,956,205	37%	652	417	36%	1229	804	35%

Table 4.2: Comparison of ThunderBird's Placement with Industrial Placement

Table 4.3 shows the effects that the global routing optimization done by TimberWolf and the channel routing done by YACR have on the total number of tracks used. Two types of optimizations are done by TimberWolf to reduct the total channel density. First, net segments that (because of electrically equivalent pins) can be placed either above or below a block of cells are assigned to a particular channel. Second, adjacent cells are swapped and entire rows are shifted with the goal of reducing the density, instead of the wire length.

The table lists the total density of the channels before global routing optimization, the density after the positions of the switchable net segments have been decided on, the density after cell swaps and row shifts have been done, and the total number of tracks actually needed by YACR.

Circuit	Total Number of Tracks			
	Before Optimization	After Selecting Switchable Net Segments	After Row Shifts and Cell Swaps	After Routing
ind1	96	96	90	95
ind2	134	114	109	115
ind3	438	438	417	436
ind4	67	59	51	53
ind5	371	334	289	298

Table 4.3: Number of Tracks Needed

Tables 4.4 to 4.7 describe the performance of YACR when routing the channels of the industrial circuits. The channels of each example are listed, along with the number of nets and columns in each one, the density, the number of tracks needed to route them, and whether or not cycles are present in the vertical constraint graph.

Two things are of note. First, YACR was almost always able to route the channels within one track of density. Second, about half of the channels have cycles.

Channel No.	Nets	Columns	Density	Tracks Used	Cycles?
1	18	154	3	3	N
2	64	187	8	9	Y
3	66	196	10	10	Y
4	69	196	7	7	N
5	84	194	10	11	Y
6	45	190	5	5	N
7	84	185	13	13	Y
8	61	186	7	7	Y
9	76	190	15	15	N
10	55	189	5	5	Y
11	35	187	10	10	N

Table 4.4: Channels of Circuit Ind1

Channel No.	Nets	Columns	Density	Tracks Used	Cycles?
1	14	220	2	2	N
2	64	213	10	10	N
3	50	219	10	10	N
4	63	214	9	10	Y
5	87	209	9	10	Y
6	72	202	9	10	Y
7	80	205	10	10	N
8	60	204	8	9	Y
9	49	208	6	6	Y
10	57	218	7	8	Y
11	47	218	7	7	N
12	81	213	15	15	N
13	46	213	8	10	Y

Table 4.5: Channels of Circuit Ind2

Channel No.	Nets	Columns	Density	Tracks Used	Cycles?
1	565	88	10	10	N
2	591	186	13	13	Y
3	601	218	16	17	Y
4	583	212	17	17	Y
5	586	244	17	17	Y
6	614	277	24	25	Y
7	596	256	26	26	Y
8	582	247	22	23	Y
9	452	249	27	28	N
10	472	258	27	28	Y
11	621	299	32	33	N
12	630	337	32	33	Y
13	639	340	34	35	Y
14	634	335	30	31	Y
15	610	296	25	26	Y
16	620	317	25	26	Y
17	612	266	20	21	Y
18	590	202	13	14	Y
19	579	63	12	12	N

Table 4.6: Channels of Circuit Ind3

Example	Channel No.	Nets	Columns	Density	Tracks Used	Cycles?
ind4	1	76	368	8	8	N
	2	175	438	17	17	N
	3	153	441	17	18	Y
	4	47	372	10	10	N
ind5	1	61	404	12	12	N
	3	89	442	15	15	N
	4	168	448	24	24	N
	5	168	439	24	24	N
	6	192	446	38	39	Y
	7	175	436	25	26	N
	8	162	447	21	21	N
	9	197	448	26	26	Y
	10	169	448	21	22	N
	11	165	444	21	22	Y
	12	157	442	23	23	Y
	13	92	417	12	12	Y
	14	79	373	13	13	N
	15	13	320	2	2	N

Table 4.7: Channels of Circuit Ind4 and Ind5

CHAPTER 5

Conclusions

In this report, three topics have been addressed: extension of the capabilities of an existing channel router, YACR, development of a new multilayer channel router, Chameleon, and of a standard cell placement and routing system, ThunderBird.

The channel router YACR has been extended to handle a greater number of routing problems. The ability to specify precisely fixed terminals on one end of a channel has been added to YACR. This will allow YACR to be used in a system for routing L-shaped channels, which will improve the performance of custom-cell placement and routing systems.

The ability to route channels with irregular edges has also been added to YACR. Channels with irregular edges often occur in custom-cell layouts, and they exist in standard-cell layouts whenever the height of the blocks is not uniform. When a channel router that can only handle perfectly rectangular channels is used for irregular channels, it cannot take advantage of the space within the irregularities, and it uses more rows than necessary. The new version of YACR is able to use this extra space. An important point is that YACR's performance on straight channels is not compromised. This is important in an automated placement and routing system because the channels to be routed may have relatively deep indentations, shallow indentations, or no indentations at all, and the quality of the routing must be as high as possible for every situation.

Work in progress to improve the performance of YACR with irregular channels includes adding the ability to place doglegs in critical nets. This will improve the quality of the results when very deep indentations are present, and nets must 'bend' around them. We are also planning to add obstacle avoidance to YACR, and the ability to have doglegs will make this easier.

The second part of this report described the multi-layer channel router Chameleon.

Many of the concepts in YACR were generalized to handle arbitrary numbers of layers, and the maze routing (a critical part of YACR) was generalized, and completely rewritten. The performance of Chameleon is impressive; when 4 or more layers are available, the routing is almost done in the minimum possible number of rows. When using 2 or 3 layers, Chameleon performs at least as well as the best previously existing channel routers. Because of its more generalized maze routing, Chameleon is often able to outperform YACR on two-layer problems. A fundamental principle of Chameleon is that the same algorithms are used to handle any number of layers and combination of track spacings. This greatly increases its robustness, allowing good results to be obtained for bizarre or unanticipated types of routing problems.

We plan to have Chameleon replace YACR as our main production channel router. Other work in progress on Chameleon includes adding the ability to dogleg nets, and the ability to specify irregular channel edges. When these are done, Chameleon will be able to handle the widest range of channel routing problems, producing good results in every situation.

The third part of this report described the standard cell placement and routing system ThunderBird. ThunderBird integrates the TimberWolf standard cell placement program and the YACR channel router into the Hawk/Squid CAD database system used at Berkeley.

Both TimberWolf and YACR deal with symbolic data. ThunderBird must generate the symbolic problems for TimberWolf and YACR from the actual design data, and translate the symbolic solutions generated by TimberWolf and YACR to actual physical layout data. At the same time, it must allow the user to specify the design rules for the routing, and produce compact wiring without any design rule violations.

ThunderBird has several features that extend the range of design styles it can be adapted to. First, it is ungridded, and cell boundaries and terminal locations may be at arbitrary locations. To accomplish this, ThunderBird must translate the arbitrary

coordinates of the cell features to a symbolic grid that YACR uses for routing. The ability of YACR to handle irregular channels has also been exploited, and cells may have different heights. The user may also specify electrically equivalent terminals on the cells, and TimberWolf's global router will take advantage of them to minimize wire length and total channel density. Finally, the terminal locations and cell boundaries are extracted directly from the cell layouts, thus reducing the amount of design data that must be maintained.

The high-quality of the placements generated by TimberWolf and the routing done by YACR have allowed ThunderBird to produce excellent results. We have been able to reduce the area of several industrial examples by 30 to 50 percent. The high quality of the placement and routing can be easily seen in the dense, compact wiring of the channels.

A good deal of work on ThunderBird is in progress or planned. The multi-layer channel router Chameleon will be incorporated into the system. This will allow the latest double-metal IC fabrication technologies to be used. Also, ThunderBird will be used to perform the placement and routing for a random logic synthesis system under development at Berkeley. The ability of ThunderBird to place and route cells of varying heights using more than two layers of interconnect will allow the generated layouts to be as compact as possible.

References

[Bu85] Jeff Burns and Karti Mayaram, "An N-Layer Channel Router", *Unpublished Report*, EECS Deparment, Univ. of Cal., Berkeley, May 1985.

[Ch84] Y.K. Chen and M.L. Liu, "Three-Layer Channel Routing", *IEEE Trans. on Computer-Aided Design*, vol. CAD-3, pp. 156-163, 1984.

[Da85] W.-M. Dai, T. Asano, and E.S. Kuh, "Building Block Layout: Routing Region Definition and Ordering Scheme", Electronics Research Laboratory Memo #UCB/ERL M85/14, Univ. of Cal., Berkeley, Feb. 1985.

[De76] D. Deutsch, "A 'dogleg' channel router", *Proc. 13th Design Automation Conf.*, pp. 425-433, 1976.

[De76b] D. Deutsch, "LTX: A System for the Directed Automatic Design of LSI Circuits", *Proc. 13th Design Automation Conf.*, pp. 399-407, 1976.

[Ha71] A. Hashimoto and J. Stevens, "Wire Routing by Optimizing Channel Assignment", *Proc 8th Design Automation Conf.*, pp. 214-224, 1971.

[Ha85] Susanne Hambrusch, "Channel Routing Algorithms for Overlap Models", *IEEE Trans. on Computer-Aided Design*, vol. CAD-4, pp. 23-30, 1985.

[Ke84] K. Keller, "An Electric Circuit CAD Framework", Electronics Research Laboratory, University of California, Berkeley, June 1984, Memo #UCB/ERL M84/54.

[Ke84b] K. Keller, "Manual for the Squid Package", *Internal Memorandum*, CAD Group, University of California, Berkeley.

[Ki83] S. Kirkpatrick, C. Gelatt, and M. Vecci, "Optimization by Simulated Annealing", *Science*, vol. 220, n. 4598, pp. 671-680, 13 May 1983.

[Ko82] Robert K. Korn, "An Efficient Variable-Cost Maze Router", *Proc. 19th Design Automation Conf.*, pp. 425-430, 1982.

[Le61] C. Y. Lee. "An Algorithm for Path Connections and its Applications", IRE Trans. on Electronic Computers, vol. EC-10, pp. 346-365, Sept. 1961.

[Li81] M.L. Liu and Y.K. Chen, "Double Layer Channel Routing with Irregular Boundaries", Electronics Research Laboratory Memo #UCB/ERL M81/79, Univ. of Cal., Berkeley, Oct. 1981.

[Ma85] Hi Leung Ma and Srinivas Devadas, "A Two-to-Infinity Layer Channel Router", *Unpublished Report*, EECS Deparment, Univ. of Cal., Berkeley, May 1985.

[Ot82] R.H. Otten, "Automatic Floorplan Design", *Proc. 19th Design Automation Conf.*, pp. 261-267, 1982.

[Ri82] R.L. Rivest and C.M. Fiduccia, "A 'greedy' channel router", *Proc. 19th Design Automat. Conf.*, pp. 418-424, 1982.

[Ro85] Fabio Romeo and Alberto Sangiovanni-Vincentelli, "Probabilistic Hill Climbing Algorithms", *1985 Chapel Hill Conference on Very Large Scale Systems*, Computer Sciences Press, May 1985.

[Ry85] D.E. Ryan, "A Routing Toolbox for Interactive Custom IC Layout", Electronics Research Laboratory Memo #UCB/ERL M85/40, Univ. of Cal., Berkeley, May 1985.

[Sa84] A. Sangiovanni-Vincentelli, M. Santomauro, and J. Reed, "A New Gridless Channel Router: Yet Another Channel Router the Second (YACR-II)", *IEEE Trans on Computer-Aided Design*, vol. CAD-4, pp. 208-219, 1984.

[Se84] Carl Sechen and Alberto Sangiovanni-Vincentelli, "The TimberWolf Placement and Routing Package", *Proc. 1984 Custom Integrated Circuit Conf.*, 1984.

[Se85] Carl Sechen, "The TimberWolf Standard Cell Placement and Routing Program: User's Guide for Version 3.2", *Internal Memorandum*, CAD Group, University of California, Berkeley.

[So81] J. Soukup, "Circuit Layout", *Proc. IEEE*, vol. 69, no. 10, October 1981.

[Yo82] T. Yoshimura and E.S. Kuh, "Efficient Algorithms for Channel Routing", *IEEE Trans. Computer-Aided Design*, vol. CAD-1, pp. 25-35, 1982.

APPENDIX A

User's Guide to YACR

YACR User's Guide
For Version 2.1

James Reed
Douglas Braun

1. Problem Formulation

The channels that YACR can route have the following characteristics:

i) a rectangular region with no obstructions, but possibly variations in the height of two opposite sides;
ii) two layers to be used for routing, referred to in this guide as "metal" and "poly" but the actual material is unimportant;
iii) fixed terminals on two opposite edges of the rectangular region, called the "top" and "bottom" edges;
iv) (optional) terminals on the other two edges, "left" and "right", of the rectangular region, exact position of these terminals is not specified, but order may be specified for one edge, or exact locations may be specified for one edge.

All of the routing is done within the boundaries of the rectangular region. The distance between the top and bottom edges is determined by YACR, but will be minimized. If fixed end terminals are specified, the width of the channel is implicitly determined by the locations of the fixed end terminals. The length of the top and bottom edges are given by the user, but in rare cases may be extended by YACR to complete the route. The top and bottom edges will be extended only by adding empty columns to their left or right ends; the spacing between terminals on the top and bottom edges will never be altered by YACR.

It is possible to specify variations in the height of the top and bottom edges. This is done by specifying the relative distance towards the center of the channel for each used or unused terminal position on the top and bottom of the channel. When this is done, the top or bottom of the channel can be thought of as an irregular manhattan stairstep, with terminals only on the horizontal portions.

YACR is guaranteed to make all of the connections specified in its input, unless fixed end terminals are specified, in which case it will fail if the channel cannot be routed in the width implicitly specified by the side with fixed end terminals. The connections to the terminals on the top and bottom edges will be made with poly. The connections to the terminals on the left and right edges will be made with metal.

YACR routes on a symbolic manhattan grid. The metal layer is used for almost all routing segments that run parallel to the top and bottom edges; poly is used for almost all segments parallel to the left and right edges. This choice was made because the length of segments running from top to bottom is usually less than the length of segments running from left to right and metal has lower resistance than poly.

YACR is a *symbolic* channel router; it does not specify routing geometries in terms of actual dimensions or location on a chip, but shows their relative positions in the channel. The output is comparable to a stick diagram of a circuit layout.

An interface to the HAWK graphics editor and SQUID data base that translates the symbolic output of YACR into physical geometries has been developed. For information on how this interface is used see *User's Guide to Routing in HAWK*.

2. Yacr Input and Output

2.1. Yacr Input File Formats

Yacr can use either of two formats for input. The information that is supplied by both formats consists of: the number of nets to be routed; the number of columns in the channel; a list of pins that connect to the top edge of the channel; a list of pins that connect to the bottom edge of the channel; the number of pins that connect to the left edge of the channel; the list of pins on the left edge; the number of pins that connect to the right edge of the channel; and the list of pins on the right edge. Two lists of "offsets" may also be specified. The list of top offsets gives the relative distance towards the center of the channel of each top pin. The list of bottom offsets gives the relative distance towards the center of the channel of each bottom pin.

First is a description of the default format, then the format called for by the "-H" command line argument to Yacr.

The number of nets in the channel is specified with the following line:

```
nnet= #
```

The number must be separated from the equals sign by at least one white space character, a space, a tab, or a newline. No other spaces are allowed.

The number of columns in the channel is specified with the following line:

```
ncol= #
```

As with the number of nets, the only space allowed is between the equals sign and the number. The space is also required.

The list of pins on the top edge of the channel is begun with the keyword "top_list". The keyword is followed by a white space separated list of numbers, representing nets. The list must contain "ncol" integers. The numbers need not be consecutive. "0" is a special number that means "this location has no net connected to it."

The list of pins on the bottom edge of the channel is begun with the keyword "bottom_list". The restriction for the top list also apply to the bottom list.

The list of nets connecting to the left edge of the channel is begun with the keyword "left_list". The keyword is followed by a number telling how many nets there are in the left list. This is followed by the list of nets. If there are no nets connecting to the left edge, this category may be omitted. By default, the order in which the nets are listed is not meaningful. If a specific order is desired, "left_list" should be preceeded by the keyword "relative". The order in which the nets are listed is then the order (from top to bottom) in which they will appear in the final route. If the word "fixed" instead of "relative", then an exact placement of the edge nets is specified. In this case, the edge list can contain zeroes like the top and bottom lists. The number of entries in a fixed edge list must be at least equal to the inital width of the channel, which is the same as the maximum channel density. If the list is longer, the inital width of the channel will be increased to the length of the list.

The list of nets connecting to the right edge of the channel is begun with the keyword "right_list". This category is exactly like the left list category. An important restriction to note is that relative or fixed order can be specified for only the left list the right list. If it is specified for both, it will only be meaningful for the one that appears second in the input file.

The optional list of offsets for the top pins of the channel is begun with the keyword "top_offset". It is followed by the list of offsets, which must contain "ncol" integers. Each entry gives the relative distance towards the center of the channel of the corresponding pin location or channel boundary, in grid units. The offsets normally range from zero upwards (with nonzero values representing an indentation into the side of the channel), but the values are internally scaled, so only the relative values matter. If all the values are identical, the top border of the channel is straight, and the results obtained will be the same as if no offsets were specified.

The list of offsets for the bottom pins is begun with the keyword "bottom_offsets". and is similar to the list of top offsets. Note that a positive value still means that the corresponding pin or channel border is moved closer to the center of the channel. One or both of the offset lists may be omitted.

The "-H" format is identical to the default format except that all keywords are omitted (except "relative" and "fixed") and all categories must be included in the following order:

number of nets
number of columns
top list
bottom list
left list
right list
top offsets (optional)
bottom offsets (optional)

The left list and the right list must be included, if there are no nets connecting to the edge of the channel, the list will be a single "0". The file will contain nothing but integers separated by white space, unless the keyword relative is included for the left or right list. The top and bottom offsets may be omitted if the sides of the channel have no indentations.

2.2. Yacr Output File Formats

Just as there are two input formats, there are two output formats. One corresponds to the default input format and the other is used with the "-H" flag or "-O" flag. The default format is described first, then the differences between it and the -H format follows.

The file begins with the following information: input file; number of nets; number of columns; number of nets in the left list; number of nets in the right list; a list of edges removed from the vertical constraint graph to make it acyclic (if any); a list of columns with maximum density; a list of columns that appear to be the best choices for starting column; and the actual starting column.

The next section of the output contains a block for each "major attempt" Yacr makes at routing the channel. Each block has the number of rows used for the attempt, followed by a list of vertical constraint violations (VCVs) that occurred, followed by a list of how each VCV column was routed (if it was). If a row was added to complete the route without starting over, that is included at the end of the last block.

The next and most important section gives the routing of the channel. The horizontal (metal) layer is given first, followed by the vertical (poly) layer. Each layer is described by a matrix of integers. Each row in the matrix represents a column in the channel (this keeps the lines shorter so a printout is easier to read). There are rows to represent the left and right edges of the channel; the extra rows are used to indicate nets that connect to the edges. Each integer in the matrices represents the net that occupies a given space in the grid. "0" means the space is empty. If offsets were specified, spaces that correspond to indentations in the channel (where no material can be placed) are represented by "X". Vias are not explicitly given, but implied at each location that the metal layer has the same net as the poly layer.

Finally, there is a summary that tells the number of vias used in the route, the total net length of all the nets, the longest net, and how much metal and poly were used to route the longest net.

The "-H" output format contains a minimum amount of information. It contains no keywords, just integers. The first two lines have the number of rows and columns, respectively, in the channel. The next row is the top row of the metal layer of the

channel, listed from left to right. The entire metal layer is given followed by the poly layer. There are no extra columns representing the edges of the channel. The last line of the file has three integers, the first is the number of the net with the longest route, the amount of metal used to route it, and the amount of poly used to route it. Spaces inside indentations are now represented by a "0".

2.3. Examples

2.3.1. Example 1, Input (Default Format)

```
nnet= 2 ncol= 3
top_list
 2 0 1
bottom_list
 1 0 2
```

2.3.2. Example 1, Output (Default Format)

```
Input file: ex/2

num_nets= 2, num_cols= 3, num_left_nets= 0, num_right_nets= 0
Not closing cycle with edge from 1 to 2
Columns with maximum density:  1
Best choice(s) for starting column is (are):  1
initial_column = 1
num_rows = 2
VCV: nets  1  2, column   3
column 3 was not routed
Row 1 is being used to complete the route.
The final result is:

The horizontal layer (turned sideways):
 0  0  0  col =  0
 0  2  2  col =  1
 0  2  0  col =  2
 0  2  0  col =  3
 0  0  0  col =  4

The vertical layer (turned sideways):
 0  0  0   col =  0
 1  0  2   col =  1
 1  1  1   col =  2
 2  2  1   col =  3
 0  0  0   col =  4

There are 2 vias
The total net_length is 12
The longest net is 2, metal length = 4, poly length = 3
```

2.3.3. Example 2, Input (Default Format)

```
nnet= 72
ncol= 169
top_list
  1  2  4  6  8 10 11 13  3  9 16  5 17 11  5 14 14  7 12 17 19  1 20 21
```

```
 23 24  0 16 10  3 11 25  0 26 11 26 11  0 27 28 11  3  9 16 30 27  5 31
  1  5  1 20 32 23 24  0  9  1 20 29 23 24  0  3  8 30 38 28 19  6 40 27
 35 41 42  6 19 34 43 30  8 31 43 39 46 36 46 47 48 31  0 24 23 45 20  1
 51  0 40 39 40 39  0  8 30 50 54  0  0 55 49 19  6  0 47 42 47 42  0 53
 58  6 19 49 50 30  8 60 62 59 54 55 54 56 63 55 65  0 66 68 66 68  0 60
 68  0 46 44 46 44  0 69  0 55 58 55 58  0 64 71  0 72 63 72 63  0 57 62
 54
bottom_list
  0  3  5  7  9  5 12 14 15  7 12 14  7  4 13  8  6 15 18 14  8  6 11 22
 21  0 18 16 18 16  0  8  6 26 11  0 24 23 25 20  1 29  0 22  3 22  3  0
  0  9  2  9  2  0 32 23 33 19  6  8 30 27 34 35 36 37 39 31 39 35 38 31
  8 30 37 41 19  6 44 45  0 33 31 33 31  0 27 35 36 48 49 31 39 46 47 50
 52 20 53 24  0 47 39  0 24 51 20 52 20 52 23  8 30 50 56  0  0 57 49 19
  6  6 19 49 59  0  0 61 50 30  8 55  0 24 64 20 52  0 67 68 63 55 24 52
 20 69 24  0 46 62 63 68  0 24 65 20 52  0 70 60 62 54 63  0 24 71 20 52
 67
relative right_list 6
68 55 63 70 67 61
top_offsets
  0  0  0  0  1  1  1  3  3  3  3  3  3  3  3  3  3  2  2  2  2  2  2
  2  2  2  2  2  2  2  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  1
  0  0  0  0  1  1  1  3  3  3  3  3  3  3  3  3  3  2  2  2  2  2  2
  0  0  0  0  1  1  1  3  3  3  3  3  3  3  3  3  3  2  2  2  2  2  2
  2  2  2  2  2  2  2  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  1
  1  1  1  1  5  5  5  5  5  5  5  5  5  4  4  4  4  4  4  4  4  4  4
  2  2  2  2  2  2  2  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  1
  1
bottom_offsets
  2  2  2  2  2  2  2  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  1
  0  0  0  0  1  1  1  3  3  3  3  3  3  3  3  3  3  2  2  2  2  2  2
  0  0  0  0  1  1  1  3  3  3  3  3  3  3  3  3  3  2  2  2  2  2  2
  0  0  0  0  1  1  1  3  3  3  3  3  3  3  3  3  3  2  2  2  2  2  2
  2  2  2  2  2  2  2  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  1
  2  2  2  2  2  2  2  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  1
  1  1  1  1  5  5  5  5  5  5  5  5  5  4  4  4  4  4  4  4  4  4  4
  4
```

3. Running YACR

3.1. Description

YACR2 is run with the following command:

yacr [options] [file1 [file2]]

File1 contains the input, file2 gets the output. If file2 is omitted, the output goes to standard out. If no files are specified, input is read from standard input, and output goes to standard output.

The command line options described below can be specified in any order, but must come before the input and output file names.

-a Add columns at the ends of the channel if necessary to complete the route. By default columns are not added, but if the channel has cyclic constraints, additional columns may be necessary. If *yacr* cannot complete a route, it will give a message recommending that this flag be used.

-c n Begin routing the channel starting in column "n". If this is not specified, or if "n" is less than one or greater than the number of columns in the channel, *yacr* will chose a starting column. If the left_list or right_list is specified in "relative" or "fixed" order, the "-c" option will be ignored.

-d Sets the debug flag so that the channel will be printed (in the desired format) at the following times: when the current number of rows is found to be insufficient; when the route is completed, before metal maximization and cleanup; after metal maximization and cleanup. Useful only for debugging purposes.

-H Use an alternate input and output format. Also forces input to be read from standard input, and output to be written to standard output.

-O filename
Read the normal input format, but place the results in the alternate format in "filename". The statistics, etc., are still written to any output file specified in the normal way. Useful to prepare output for arrayToCif or other post-processors.

-s Do not generate output for the channel; only print statistics for the channel width, number of vias, etc.

-C Do not print the list of cycles in the vertical constraint graph. Channels with many fixed end pins may have thousands of cycles.

-m Do not perform metal maximization.

3.2. Diagnostics

The following error messages are all written to stderr. Yacr will return a nonzero exit status if there is any fatal error.

"net n is not properly routed
bad pins are: col1/edge1 . . . "
The verification routine has discovered a problem with the route. The pin of net "n" in column "col1" on edge "edge1" was not connected to the leftmost pin of net "n". This message will only appear when a bug in the code manifests itself.

"The channel cannot be routed without additional columns,
please use the "-a" command line argument."
The channel has cyclic constraints that cannot be routed in the space provided. This has not happened on any of the hundreds of industrial channels routed by YACR to date.

"top_list already specified, first list used."
The input file tries to specify the top_list portion of the input more than once, all lists after the first are ignored. There are similar messages for the bottom_list, left_list, right_list, nnet, and ncol.

3.3. Suggestions for Best Results

The best suggestion that can be given here is: Let YACR make as many decisions as possible. There are two ways in which the user may make decisions for YACR, either by specifying that the left_list or right_list is "relative" or "fixed", or by specifying the starting column with the "-c" command line argument. One decision that the user should make for YACR is whether or not it should be allowed to add columns to the channel ends to complete the route.

Relative Order

If the left_list or right_list is given a specfic order YACR will almost always require more rows for routing that if the order was not given. This happens for two reasons: first, there are more restrictions that must be met by the final route; and second, in order to guarantee the restrictions are met, YACR must begin routing at either the left or right

end of the channel (usually a bad place to start). If a fixed left or right list is given, there will be even more restrictions. Also, if there are fixed end pins, the channel width will not be automatically expanded if Yacr fails to route. In this case, it is necessary to restart the problem, with another row specified by placing it in the list of fixed end pins. Remember that the list of fixed end pins must be at least as large as density.

Use of -c

Most of the time that YACR is allowed to choose its starting column it will route a channel in density. Occasionally, a small number of extra rows will be required to complete the route. Since the number of rows needed for the route will vary by one or two depending on which column YACR started with, the user can sometimes get YACR to route a channel in density by forcing it to begin in a column it would not normally choose.

Note that if the pins on the left or right edge have a specified order, the user is already forcing YACR to start in the first or last column.

The YACR output file contains information that is useful in picking a starting column that might be better than the column YACR chose. Consider the following portion of a YACR output file (only the routing of the channel is omitted):

```
Input file: ex/3c
num_nets= 54, num_cols= 103, num_left_nets= 0, num_right_nets= 0
Columns with maximum density:  58 69
Best choice(s) for starting column is (are):  58 69
initial_column = 58
num_rows = 18
VCV: nets 38 45, column  71
VCV: nets 21 45, column  63
VCV: nets 21 24, column  75
column 75 was routed by maze1a
column 63 was not routed
column 71 was not routed
Row 1 is being used to complete the route.
```

In this example YACR was allowed to choose a starting column. It chose column 58, and ended up with two vertical constraint violations (VCVs) (in columns 63 and 71) that it could not remove. We could hope for better results by forcing YACR to start in any of columns 69, 63, or 71. We might choose 69 because YACR said it might be a good choice, or we might choose 63 or 71 in hopes that a difficult-to-remove VCV would be avoided. If we choose to force YACR to start routing in column 63 (with the command "yacr -c 63 inputfile outputfile") we get the following results.

```
Input file: ex/3c
num_nets= 54, num_cols= 103, num_left_nets= 0, num_right_nets= 0
Columns with maximum density:  58 59 69 70
initial_column = 63
num_rows = 18
VCV: nets 41 26, column  72
VCV: nets 49 35, column  90
column 72 was routed by maze2
column 90 was routed by maze2
```

With the forced starting column YACR was able to route the channel in density.

There are several interesting things should be pointed out about the two output files. Notice that the first one lists 58 and 69 as the columns with maximum density and the second lists colums 58, 59, 69, and 70. This is not a contradiction, in this example columns 58 and 59 are crossed by the same nets (the same is true of columns 69 and 70). If YACR began routing in either column 58 or 59, it would achieve exactly the same

results, so it removes all but one of the redundant columns from consideration. When the user specifies a starting column, YACR does not waste time figuring out if adjacent columns of maximum density are identical.

The other interesting fact is that there were not only a different number of columns with vertical constraint violations, but the columns were totally different when YACR began routing in a different column.

Irregular Channels

Yacr can handle channels with irregular sides. This is done by specifying the top and bottom offset lists in the input. Some points must be remembered. First, the offsets apply to the places where there are no pins, as well as the columns with pins. Second, if fixed end pins are specified at the same time as irregular edges, the horizontal segments of the fixed pins must not run into the indentations. Yacr will fail if this happens. Last, Yacr was only designed to handle relatively small indentations. If the effect of the indentations is to give the channel a zig-zag shape, and there are nets that span a large portion of the channel, Yacr will use more tracks than necessary. This is because Yacr does not dogleg nets, so the nets cannot bend to get around the indentations.

Use of -a

There are some cases in which YACR will think it needs extra columns, but could actually complete the route by adding only rows. Since the times when YACR really needs extra rows are rare (it has not yet happend in a industrial example), you should only use the "-a" command line argument after YACR tells you it is necessary.

3.4. Reed's Rule of Routing

Routers do weird things.

When looking at the output of YACR (or any other router) it is always a good idea to keep in mind this variant of the widely known Murphy's law.

APPENDIX B

User's Guide to Chameleon

CHAMELEON User's Guide

Douglas Braun

Chameleon is a channel router that is capable of using any number of layers for routing greater than one. The track spacing of each layer may be independently specified. Chameleon was derived from the channel router YACR, so its operation is similar. If you are planning to use Chameleon on a regular basis, it might be a good idea to read the *YACR User's Guide* and/or the cadman pages for YACR (YACR(CAD1) and YACR(CAD5)). This guide assumes that you are familiar with the concepts of channel routing. Doug Braun's master's report has much more information on the concepts of multi-layer channel routing and the theory of operation of Chameleon.

1. Invoking Chameleon

Chameleon is invoked as follows:

chameleon [-s] [-d] [*inputfile* [*outputfile*]]

The -s option suppresses the printing of the routed channel; only the statistics and diagnostics are printed. The -d option prints a very large amount of data useful only for debugging. Since much of the code of Chameleon was stolen from YACR, there may be other options that have an effect. However, nothing else but -s or -d is guaranteed to work. If *inputfile* or *outputfile* are not specified, standard input and output are used instead.

Two other files must be present for Chameleon to work. First there must be a executable copy of the partitioning program *part*. Since the *system()* library routine is used to automatically invoke the partitioner, the executable file must be located in a directory specified by your environment's **$PATH** variable. If only 2 or 3 layers are being used, no partitioning is done, and the partitioner will not be invoked. The other file is a short text file that contains the specifications for the number of layers and the width and spacing of the material for each layer. It must be in the current directory, and be named **.masks**. An additional temporary file is generated, whose name is "partfile". It may be removed after each run.

2. Input File Format

The format of the input file is compatible with the human-readable format of YACR. The only differences are that **relative** or **fixed** end pins cannot be specified, and top or bottom offset lists are ignored. Note that there is only one format for the input file (there is no counterpart of the YACR "-H" format).

The information that is supplied in the input file consists of: the number of nets to be routed; the number of columns in the channel; a list of pins that connect to the top edge of the channel; a list of pins that connect to the bottom edge of the channel; the number of pins that connect to the left edge of the channel; the list of pins on the left edge; the number of pins that connect to the right edge of the channel; and the list of pins on the right edge.

The number of nets in the channel is specified with the following line:

nnet= *number*

The number must be separated from the equals sign by at least one white space character, a space, a tab, or a newline. No other spaces are allowed.

The number of columns in the channel is specified with the following line:

ncol= *number*

As with the number of nets, the only space allowed is between the equals sign and the number. The space is also required.

The list of pins on the top edge of the channel is begun with the keyword "top_list". The keyword is followed by a white space separated list of numbers, representing nets. The list must contain "ncol" integers. The numbers need not be consecutive. "0" is a special number that means "this location has no net connected to it."

The list of pins on the bottom edge of the channel is begun with the keyword "bottom_list". The restriction for the top list also apply to the bottom list.

The list of nets connecting to the left edge of the channel is begun with the keyword "left_list". The keyword is followed by a number telling how many nets there are in the left list. This is followed by the list of nets. If there are no nets connecting to the left edge, this category may be omitted. By default, the order in which the nets are listed is not meaningful.

The list of nets connecting to the right edge of the channel is begun with the keyword "right_list". This category is exactly like the left list category.

3. The .masks File

The .masks file contains the number of layers used, and the width and self-separation of the material for each layer. A sample entry is shown here:

```
layers 3
1 1
1 2
2 3
```

The first line gives the number of layers, and should be formatted exactly as shown. Each succeeding line has the width of one layer followed by its self-separation. The width specified for each layer should account for the width of the contacts of the layer, so that contacts on adjacent tracks will be sufficiently separated. The data for layer 1 is on line 2, for layer 2 on line 3, etc. There must be one of these lines for each layer. The units are dimensionless, and are used to establish the symbolic grid that Chameleon uses for describing the routing region. Note that the track spacing (or *pitch*) of each layer is equal to the sum of its width and separation. If the format of this file is not correct, Chameleon may react in possibly unexpected ways.

Note that the data in this file should not necessarily be given in microns or lambda. Chameleon will run much faster if the data are scaled so that the width and spacing of the layer with the smallest features are both one. See the section on the grid below for more details.

4. The Output File

The format of the output file is similar, but not identical to that of YACR. Also, there is only one format, which is designed to be human instead of machine readable. A sample file is shown below.

```
Input file: test

num_nets= 4, num_cols= 9, num_left_nets= 0, num_right_nets= 0
Columns with maximum density:  5 6
Best choice(s) for starting column is (are):
max density = 3
initial_column = 5
0 edges were removed.
The vertical constraint graph contains the following cycles:
1 -> 2 -> 1
2 -> 3 -> 2
Removing edge from net 2 to net 1
Affected columns are:  2
Removing edge from net 3 to net 2
Affected columns are:  4
2 cycles were detected.
Net 2 has 1 VCVs.
Net 1 has 1 VCVs.
Column 2 was not routed by stage 1
Column 9 was not routed by stage 1
Column 2 routed by stage 2b
Column 9 routed by stage 3b
RESULTS: 3 rows  Net length: 49
The final result is:


Layer number 1 (turned sideways):
    |   .   .   .  |     col =  0
  2 |   2   1   1  |   1 col =  1
  1 |   1   1   2  |   2 col =  2
  0 |   2   2   2  |   0 col =  3
  2 |   2   3   3  |   3 col =  4
  4 |   4   4   4  |   0 col =  5
  0 |   .   .   4  |   4 col =  6
  2 |   2   2   2  |   0 col =  7
  2 |   2   3   3  |   3 col =  8
  3 |   3   3   2  |   2 col =  9
    |   .   .   .  |     col = 10
```

```
Contacts between layer number 1 and 2 (turned sideways):
    |   .   .   .   |      col =   0
  2 |   X   X   .   |    1 col =   1
  1 |   .   X   X   |    2 col =   2
  0 |   X   .   X   |    0 col =   3
  2 |   X   X   .   |    3 col =   4
  4 |   .   .   X   |    0 col =   5
  0 |   .   .   X   |    4 col =   6
  2 |   X   .   X   |    0 col =   7
  2 |   X   X   .   |    3 col =   8
  3 |   .   X   X   |    2 col =   9
    |   .   .   .   |      col =  10


Layer number 2 (turned sideways):
    |   .   .   .   |      col =   0
  2 |   2   1   .   |    1 col =   1
  1 |   2   1   2   |    2 col =   2
  0 |   2   .   2   |    0 col =   3
  2 |   2   3   .   |    3 col =   4
  4 |   2   3   4   |    0 col =   5
  0 |   2   3   4   |    4 col =   6
  2 |   2   3   2   |    0 col =   7
  2 |   2   3   2   |    3 col =   8
  3 |   2   3   2   |    2 col =   9
    |   .   .   .   |      col =  10
The total net_length is 49
The longest net is 2, metal length = 14, poly length = 11

We used 3 rows.
```

The first part of the file (up to the actual description of the channel) is very similar to YACR's output. Information is given about the size of the input, the density, and any cycles an the vertical constraint graph. As nets are placed in the channel, any VCVs created are noted, and when the maze routing is done, its results are output. If chameleon fails to complete the maze routing, and cannot route the channel without adding another row, this is noted, and the routing is re-attempted with another row added to the channel. If there are any strange problems or verification failures, these will be noted. In some cases, error messages or diagnostics may be written to the standard error output as well as the output file.

The first part is followed by a description of the contents of the routed channel, if the -s option was not given. This part contains several sections. There is one section for each layer, and one for the contacts between each pair of layers. Therefore, if there are N layers, there will be $2N - 1$ sections describing the channel.

Since the entire channel is described by a grid, each section describes the contents of every grid cell for a layer (or for an imaginary contact layer located between each pair of real layers). Each layer is printed sideways.

Each line of output contains a description of one column of the grid. First the net connected to the bottom of the column is given, followed by a "|". Then the net that the material in each grid cell belongs to is given, starting at the bottom row. If a cell contains no material, a "." is output. In the sections describing the locations of contacts, an "X" is given to signify a contact. After the number of the net of the top cell is printed, another

"T" is printed, followed by the net connected to the top of the column. Finally the column number is output.

5. The Grid

Like YACR, Chameleon describes the routing region with a symbolic, grid. However, the presence of layers with different spacings complicates things. The grid units in the horizontal direction are not based on the track spacings of the layers. Instead, there is one column of the grid for each pin position, and it is assumed that the pins are spaced far enough apart for a wire on any layer to run vertically in one column.

The grid units in the vertical direction are based on the track spacing of the layers. Internally, Chameleon pays attention only to the track spacing of a layer, not its width and self-separation. This means that a layer with a width of 2 and a separation of 3 is the same as a layer with a width of 3 and a separation of 2 (both have a track spacing of 5). The width of each layer is used to optimize the partioning, but does not affect the correctness of the results. A wire is represented internally and in the output file as having a width of one.

The grid units are derived from the track spacings as follows. If the track spacings are all even, each one is divided by two. Then one grid unit is equal to one unit of track spacing.

Chameleon will run fastest if the track spacing of each layer is the same as one grid unit. This is the same scenario used by YACR. When two layers are used, it is best to have the track spacing of each layer equal to one grid unit (in a vertical direction). For this, the .masks file would read as follows:

```
layers 2
1 1
1 1
```

When more than two layers are used, and they have different spacings, it is a good idea to scale things so that the width and separation of the layer with the smallest feature size (usually poly) are both one, and the track spacings of the other layers (the sum of the width and separation) are all even. This will allow the grid units to be as large as possible.

If this sounds confusing, it is. The best way to understand what is going on is to run a small test case through Chameleon, using different numbers of layers and different spacings for each layer, and look at the results.

6. Miscellaneous Trivia

Although the exact width and separation of each layer can be specified, it is important to remember that Chameleon is a *symbolic* channel router. That is, the layouts it produces are abstract, and do not exactly represent the actual physical layout.

The layer that is used to make the actual connection to a terminal cannot be specified by the user. Chameleon will use the most convenient layer for each terminal to make the connection.

Internally, Chameleon uses each layer as either a *horizontal layer* or a *vertical layer*. These are similar to the horizontal and vertical layers of a two-layer channel router, such as YACR. A horizontal layer primarily contains horizontal segments of wiring, and a vertical layer contains primarily vertical segments. Chameleon decides which layers will be horizontal layers, and which will be vertical. It tries to maximize the number of horizontal layers, which will result in the narrowest possible channels.

Chameleon enforces a rule that says contacts may not be stacked. That is, no more than one contact may exist at any grid location. If your technology allows contacts to be stacked, do not worry about this, because enforcing the rule rarely hurts the the quality of

the results.

Chameleon verifies its results, and if an internal error results in an incorrect result, appropriate messages will be output.

APPENDIX C

User's Guide to Flounder

User's Guide to FLOUNDER

Douglas Braun

1. Overview

Flounder is a program that generates a cell list from a Squid view suitable for input to TimberWolf. Flounder is part of the ThunderBird system for standard-cell placement and routing.

Flounder extracts two kinds of data from Squid. First is the cell list and connectivity of the circuit. Second is the dimensions and pin locations of the cells used in the circuit. The output is a text file to be used as the ".cel" file that TimberWolf expects to see.

This user's guide assumes that you are familiar with the Squid/Hawk environment, and also somewhat familiar with the operation of TimberWolf.

2. Invoking Flounder

Flounder is invoked as follows:

flounder **circuitname** [**masterview** [**instview**]]

When invoked with only one argument, flounder will read a schematic view of **circuitname**, and produce a file in the current directory named **circuitname.cel**. If a view other than schematic must be read, the second argument will specify it. If there is a third argument, it will specify the view of the instances to be examined. This will be explained below.

3. The Connectivity

Flounder extracts the connectivity of the circuit (which can be heirarchical) from the Squid view specified on the command line. This information is contained in the Squid views in the form of *nets*, *terminals*, and *instances*. It must be stressed that nets and terminals are *abstract*. When you look at a Squid view with Hawk, you don't see them. What you see are pieces of geometry that *implement* them. If the view containing the circuit is essentially a netlist, it would appear empty when viewed with Hawk, because there would be no geometries to be displayed.

Flounder would be perfectly happy with such a view, but it makes some assumptions (which can be overridden by the command-line arguments specified above) that optimize it for use with *schematic* views. Graphically, a schematic view is a schematic diagram. The actual connectivity information is established by the nets, terminals, and instances it contains. However, it also contains lines, boxes, etc. (which implement the nets and terminals) that allow you to see what the circuit looks like.

Hawk has tools (crummy tools) to allow a user to create a schematic diagram at the terminal. These automatically make sure what you draw on the screen actually reflects the connectivity established by the nets, terminals, and instances.

4. Hierarchy

Flounder will accept *hierarchical* views, and flatten the hierarchy, as required by TimberWolf. The ability to have hierarchical schematics is especially useful for standard-cell design, because circuits may be composed of thousands of standard cells. The hierarchy allows the schematic diagram for some subcell to be created independently of the main top-level schematic. Flounder will find all the schematics that make up the circuit, and combine them to produce its output.

Hierarchy in schematics is done a little differently than in physical (layout) views. A physical view of a chip will contain instances of other physical views. These views will in turn contain instances of still other physical views, etc. etc. This is made painfully obvious when Hawk pushes the entire hierarchy of a chip to display its layout. Schematic views do not contain instances of other schematic views. Instead, they contain instances of *body* views. A body view is simply a graphic icon that represents the sub-circuit defined by its schematic view. For example, the body view of an AND gate cell would probably be the familiar rectangle with one side replaced by a semicircle. The body view will contain terminals (represented by little dots or boxes) that correspond to the actual terminals of the circuit it represents.

Flounder's basic algorithm for traversing the circuit hierarchy is as follows. Flounder searches the current schematic view for body views. For every body view it finds, it sees if there is a corresponding schematic view. If it finds it, it recursively continues the traversal with the newly found schematic view. If a schematic view cannot be found, it concludes that it has reached an instance that represents a standard cell (one of the leaves of the hierarchy tree), instead of another level of the schematics. If any schematic view contains the boolean-valued property **TB_STD_CELL**, Flounder will ignore the schematic, and assume that the instance actually represents a standard cell.

This should clarify what the last two command line arguments mean. The default value for **masterview** is "schematic", and the default value for **instview** is "body". If schematics were not being used, a user might choose to have "netlist" views to hold the connectivity. Each netlist view could contain instances of other netlist views, and so on, in the same way that the physical hierarchy is done. In this case, Flounder would be invoked as:

flounder **circuitname** netlist netlist

5. The Cells

As mentioned above, Flounder reaches the bottom of the hierarchy when it cannot find a schematic view corresponding to an instance of a body view. At this point, it concludes that it has reached a representation of an actual standard cell, instead of another level of schematics. Therefore, it now looks for a **tbird** view of the instance to actually obtain the physical parameters of the cell. A tbird view is simply the actual layout of the cell, with a few things added that Flounder and ThunderBird need to know. This information does not affect the usefulness of the cell for layout; in fact, the save tbird view will actually be placed in the final layout by Termite (another part of ThunderBird). The tbird views of the cells will presumably reside in a cell library, which will be found by Squid usings its path mechanism.

5.1. The Cell's Border

Flounder needs to find out two things about the cell, its dimensions, and the locations of each of its pins. TimberWolf needs to know the height and width of each cell. Each cell is assumed to be a rectangle, so its borders are determined by its height and width. (Actually, TimberWolf needs four coordinates, but they are not independent, and can be derived from the height and width.) Since the standard cells may be designed to overlap, or not have material all the way out to their borders, the bounding box of the cell's geometries cannot be reliably used as the cell border. It is necessary to mark on the cells where the actual boundary is. Flounder deals with this in the same way that the tiling program **tpack** does: The user places a special rectangle on the layout that defines the cell's border. For Flounder, this rectangle must be on the layer **SYMBOL**. With the current incarnation of Hawk, this should appear as a unfilled yellow rectangle on the display (at least for the mosisCMOS-PW process). If there is more than one rectangle, the current version of Flounder will use the first one it finds. The rectangle may be at

location with respect to the coordinate system of the view.

5.2. The Terminals

TimberWolf needs to know the location of the terminals of each cell. Terminals are placed on the tbird view of the cell by placing rectangles of material that *implement* the terminal. Hawk has a command especially for drawing terminals. On the display, these rectangles appear in a slightly different shade than rectangles that do not implement terminals.

For each terminal that is part of the circuit, Flounder will search the tbird view for rectangles that implement it. If exactly one is found, its center determines the location of the terminal. If there is more than one, then the rest define *electrically equivalent* terminals to TimberWolf. It is a fatal error to have no rectangles implementing a terminal. The above convention for representing terminals is standard practice in Hawk and Squid. The current implementation of Termite (which performs the routing of the circuit) will make all connections to the cells with one type of material, which may be specified by the user. Generally, this would be poly (for a single-metal process). See the *Termite User's Guide* for more information on the routing and design rules.

6. Things That Will Cause Problems

If any of the schematic, body, or tbird views that form part of the circuit are missing, Flounder will fail.

There must be a correspondence between the terminals in the schematic, body, and tbird views. If at any point, Flounder cannot find a terminal that is supposed to exist, it will fail. There must be at least one rectangle implementing each terminal in the tbird views.

If the rectangle defining the cell border is missing for any cell, Flounder will fail. If there is more than one for any cell, the outcome will be undefined. If the rectangle is malformed (one side having a width of zero), this meaningless information will be passed on unchanged to TimberWolf. Output Conventions

This section assumes that you are familiar with the format of TimberWolf's input data.

The names of the cells in the output file for TimberWolf will simply be the Squid name of the cell. Therefore, the names will not necessarily be unique, since one cell may be used many times in the circuit. The cell number will be an arbitrary unique integer.

Likewise, the terminal names will be used unchanged from the Squid views. Electrically equivalent terminals will be given names of the form terminalname.*nn*, where *nn* is an integer ranging from 1 upwards.

The units used for cell dimensions and pin locations are half-lambda. Since there are 20 Squid units per lambda, the coordinates returned by Squid are all divided by 10. Hawk users should note that the default grid spacing is 20 Squid units, or one lambda. When laying out the cells, it is a good idea to make the height and width multiples of two lambda.

The net names are of the form net*nnn*, where *nnn* is an arbitrary unique integer. The Squid net names cannot be used for two reasons. First, because of the hierarchy, a net in some view deep in the hierarchy may exist many times in the circuit. Second, a net output to TimberWolf may actually consist of several different Squid nets at different levels of the hierarchy that are merged when the hierarchy is flattened.

7. Sins of Flounder

If there is a problem in the circuit data maintained by Squid, such as terminal inconsistencies, or missing views, Flounder may deliberately coredump. If this happens, and it

isn't obvious why, let someone familiar with the source code have a look at the coredump. Internal Squid errors will generally cause coredumps.

Since the center of a terminal rectangle is used to determine its location, the rectangles will either have to be of height 0 or straddle the cell border to make the terminal be located exactly on the cell border. Unfortunately, the cell border rectangle (on the SYMBOL layer) tends to obscure terminals of zero height.

8. Things that are Missing

Right now, there is no way to specify that a net is connected to a pad or macro block. Pads and macro blocks are different from cells because their location on the chip must be specified. Similarly, there is no way to specify an initial location for a cell, or that a cell is not to be mirrored. Pads and macro cells will have to be manually added to the .cel file output by Flounder.

Flounder does not understand Squid global nets. There is no way to specify built-in feedthroughs in cells.

Flounder only generates the **.cel** file. The other files that TimberWolf needs, the **.net**, **.blk**, and the **.par** files, will have to be manually created. The user will have to estimate the number of blocks needed, as well as their initial separation.

It would be easy for Flounder to automatically add net weights to the .net file by looking for properties attached to the nets.

If a tbird view of a cell has the string-valued property **TB_CELL_TYPE** set to **macro** or *pad*, Flounder will ignore the cell. If the value of the property is **stdcell**, or if the property does not exist, it will be processed normally. This will allow Flounder to be expanded to correctly deal with pads and macro blocks.

APPENDIX D

Notes on Running TimberWolf with ThunderBird

Notes on Running TimberWolf with ThunderBird

Douglas Braun

TimberWolf needs four input files to operate. This manual tells how to prepare them when using ThunderBird. It is assumed that you are familiar with the *TimberWolf User's Guide for Version 3.2*, written by Carl Sechen. To get an overview of ThunderBird, read chapter 4 of Doug Braun's master's report.

1. The .cel (cell) File

The program Flounder automatically generates the .cel file. The operation of Flounder is described in the *User's Guide to Flounder*. The current implementation of Flounder leaves out a few things that may have to be added with *vi*. Most importantly, pads and macro blocks are not automatically included, and these will have to added by hand.

2. The .blk (block) File

The .blk file must be generated by hand. Fortunately this is easy to do, because some limitations of Flounder make the contents almost trivial. First of all, the class concept is not supported by Flounder, because the initial placement of the blocks cannot be specified. The class of every block must be 1. The block heights should all be the same, and set to the height of the shortest cell used by the circuit. Remember that Flounder outputs its units of half-lambdas, where a lambda is 20 Squid units. It is OK to mirror blocks.

A sample .blk file is shown below:

```
block height 120 class 1
block height 120 class 1
block height 120 class 1
block height 120 class 1
block height 120 class 1
block height 120 class 1
```

3. The .net (net) File

The .net file should be prepared according to the instructions in the *TimberWolf User's Guide* Since Flounder renames the nets in the circuit, it would probably not be worth manually weighting individual nets. Therefore, the file should consist of one line like:

allnets HVweights [*float*] [*float*]

as described in the User's Guide.

4. The .par (parameter) File

The *TimberWolf User's Guide* should be read carefully before preparing the .par file. Many of the parameters, such as **att.per.cell**, are not affected by ThunderBird in any way. The parameters that are affected are described below.

4.1. rowSep

rowSep tells TimberWolf how far apart the rows are to be. When setting this, remember that it is relative to the block height, which is supposed to be the height of the *shortest* cell used. This is only used to optimize TimberWolf, because the Termite program

(the last part of ThunderBird) will pack the blocks as close together as possible after the routing is done. When in doubt, a value of 1.0 for small circuits, and 2.0 for big ones, will do.

4.2. invertBlocks and mergeBlocks

It is a no-no to invert or merge blocks. ThunderBird can only deal with a single group of parallel, horizontal blocks. To not use these keywords. Also, TimberWolf will not do global routing if these are used.

4.3. addFeeds and feedThruWidth

addFeeds should be used. Without this, the global routing will not make much sense. **FeedThroughWidth** should be set to the width of the **twfeed** cell in half-lambda. See the Termite User's Guide for information on constructing the feedthrough cell.

5. do.global.route, do.global.route.cell.swaps, and do.global.route.row.shifts

Do.global.route should always be used. Without global routing, Termite cannot route the circuit. The other parameters may be used if you feel like it. See the TimberWolf User's Guide for more details on these.

APPENDIX E

User's Guide to Termite

User's Guide to TERMITE

Douglas Braun

1. Overview

Termite is a program that uses the results of a TimberWolf standard-cell placement to perform the detailed routing and produce a physical layout. Termite is part of the ThunderBird system for standard-cell placement and routing.

Termite reads the data in the **.pl1**, **.pl2**, and **.pin** files output by TimberWolf. It uses the information in them to place all the cells in the right locations, set up the channel routing problems to be automatically routed by YACR, and to place the wiring in the channels between the blocks of cells.

This user's guide assumes that you are familiar with the Squid/Hawk environment, and also familiar with the operation of the other parts of ThunderBird: Flounder and TimberWolf. Termite performs the last stage of the placement and routing process. If you have successfully gotten TimberWolf to run, you are ready to use Termite.

2. Invoking Termite

Termite is invoked as follows:

termite **circuitname**

The files **circuitname.pl1**, **circuitname.pl2**, and **circuitname.pin**, produced by TimberWolf, must be present in the current directory. The result of the run will be a log file **circuitname.log**, and a Squid **tbird** view of **circuitname**. Some other temporary files will be generated.

A executable copy of **yacr**, version 2.1, must be present for Termite to *exec* during its operation. Termite uses the *system()* library routine to execute Yacr, so your shell's path must include a directory with the right version of Yacr.

3. Input to Termite

The three files produced by TimberWolf mentioned above are the main source of data. the **.pl1** file contains the final positions of every cell in the circuit. The **.pl2** file contains the location of each block of cells. The **.pin** file contains the results of the global routing, expressed as a list of every terminal in the circuit, and the net segments they are connected to.

The **.cadrc** file, described in the next section, is also read by Termite.

Termite must also be able to find the tbird views of the cells used in the circuit. These are the same views described in the User's guide to Flounder. If you have gotten Flounder to run, there should be no problem with this.

Two other tbird views must be accessible. One is the feedthrough cell inserted in the blocks of standard cells by TimberWolf. The other is a contact cell used when creating the wiring. The default names of these cells are "twfeed" and "contact", but these may be changed by appropriate entries in the .cadrc file. These cells will be described in more detail below.

4. Output of Termite

The principal result of the execution of Termite is a tbird view of the routed cells. If no Squid file for it already exists, it will be created in the current directory.

The tbird view of the routed circuit actually contains instances of several sub-views. These are stored in the same Squid directory as the circuit. These views are named

block*nn* and chan*nn*, where *nn* is an integer running from 1 upwards. Block 1 and channel 1 are at the bottom of the routed circuit, and there is one more channel than the number of blocks.

The other important file is the log file, which is described below.

Other files which are output are the channel router temporary files. They are named route.in.*nn*, and route.out, where *nn* is the channel number. Route.out is reused for each channel, and may be removed after Termite is finished. The route.in files are the channel routing problems for each channel, in human-readable Squid format, and you may want to save them for performance comparision.

5. The .cadrc File and the Design Rules

The **.cadrc** file is a central repository for technology-specific paramaters of a number of Berkeley CAD programs. A new section must be added to your version of it for Termite. If you do not have your own .cadrc file in your home directory, copy ~cad/lib/.cadrc to there, or get one from someone else who has successfully run Termite.

A sample entry is shown below:

```
#values for mosisCMOS-PW process
begin TERMITE
 NAME LAYER1_NAME      "CP"
 NAME LAYER2_NAME      "CM"
 NAME FEED_NAME       "cmosfeed"
 NAME CONT_NAME       "cmoscontact"
 VALUE CONT_HEIGHT    7
 VALUE CONT_WIDTH     7
 VALUE LAYER1_SPACING 3
 VALUE LAYER2_SPACING 4
 VALUE LAYER1_WIDTH 3
 VALUE LAYER2_WIDTH 3
 VALUE END_CLEARANCE 4
 VALUE SIDE_CLEARANCE 4
end
```

The lines have the following meaning. A line beginning with a "#" is a comment. The line **begin TERMITE** signals the start of the Termite section of the .cadrc file. All lines between this line and **end** will be read in, and the rest ignored. The rest of the lines all begin with **NAME** or **VALUE**, followed by a keyord and an argument. **NAME** means that a string-valued argument is expected, while **VALUE** means that an integer is expected. All values that represent a dimension are in lambda The order of the lines is not important.

LAYER1_NAME and **LAYER2_NAME** identify the layers to be used for the wiring. Layer 1 is the "vertical layer", and is primarily used for the vertical segments. All connections to the terminals of the cells will be made with this layer. Layer 2 is used for the horizontal segments, and the terminals at the ends of the channels will be on this layer. Layer 1 would typically be polysilicon, while layer 2 would be metal. Ideally, metal would be used for layer 2, because the long runs of wiring will be made using this layer.

FEED_NAME and **CONT_NAME** specify the names of the Squid views to be used for the contacts and feedthroughs. The type of the views is **tbird**. These are optional, and if they are omitted, the default names "twfeed" and "contact" will be used. These are explained more in the next section.

CONT_HEIGHT and **CONT_WIDTH** must be specified. They give the height and width, in lambda, of the contact. If there are bizarre design rules, these values may have to be different than the actual size of the contact to prevent spacing violations.

LAYER1_WIDTH and **LAYER1_SPACING** give the minimum width and self-separation of the material used for layer 1, in lambda. The data for layer 2 is specified the same way. All four values must be given. It is assumed that the separation between a contact and any other material is equal to the maximum of **LAYER1_SPACING** and **LAYER2_SPACING**.

END_CLEARANCE and **SIDE_CLEARANCE** give the spacing in lambda between the standard cells and any wiring. **SIDE_CLEARANCE** says how much room to allow above and below the cells. Obviously, the wires that actually connect to terminals will violate this, but they will be brought in from directly above or below the terminal location. **END_CLEARANCE** gives the required spacing between any wiring and the left and right sides of the cells. If all the cells are the same height, this parameter is meaningless. Only if cells have different heights will there be any exposed ends for wires to get too close to. The two clearance parameters are optional, and if omitted, the maximum of **LAYER1_SPACING** and **LAYER2_SPACING** will be used.

Some other parameters are not shown above, and will be rerely needed. **GRIDHEIGHT** and **GRIDWIDTH** are the dimensions of the symbolic grid used for the routing. Normally, the grid width is equal to **CONT_HEIGHT** plus the maximum of **LAYER1_SPACING** and **LAYER2_SPACING**. This is the smallest distance that can be guaranteed not to acuse any design rule violations. With some design rules, it may be necessary to adjust this. If Termite is unable to assign every terminal on a channel to a grid location, it may have to be changed. If the terminals of the cells are designed to always fall on a particular grid spacing, **GRIDWIDTH**, should be set to this value.

GRIDHEIGHT is less likely to need adjustment. The grid height is normally calculated silmilarly to the grid width, but the height will be reduced for rows of routing where contact-contact spacing is not a problem.

The parameter *NPUT_SCALE* should not need to be adjusted when Termite is run as part of ThunderBird. It determines the scaling between the data read from TimberWolf and the internal units that Termite uses. Termite's internal units are the same as Squid's units, which are 20 units per lambda. Since the data from TimberWolf is assumed to be in half-lambda, all coordinates read from TimberWolf are multiplied by 10. **INPUT_SCALE**, if specified, is the new scale factor.

6. The Feedthrough and Contact Cells

The feedthrough cell is inserted into the circuit by TimberWolf. It is treated as a cell with one terminal on top and another terminal on bottom. TimberWolf makes the following assumptions about this cell. Its width is specified by the feedThruWidth parameter in the **.par** file used by TimberWolf. Its width is specified in half-lambda. The height of the cell is assumed by TimberWolf to be the same as the block height. Things work best when the feedthrough is the same height as the shortest cell used in the circuit. This is why *Notes on Running TimberWolf with ThunderBird* says that the block height should be the same as the height of the shortest cell in the circuit. TimberWolf also assumes that the terminals of the feedthrough are centered on the top and bottom edges of the cell.

With these constraints in mind, it should not be too difficult to design the cell. You do not have to mark the locations of the terminals, but you **must** place a bounding box around the cell on the Squid layer SYMBOL. This is the same as the box defining the standard cells' dimensions, and is described in *The Flounder User's Guide*.. As mentioned above in the section describing the .cadrc file, the default name of the feedthrough cell is "twfeed". It is of course a third view, like the views of the rest of the cells.

The contact cell is a tbird view of the contact between the two layers used for routing. Its width must be the same as specified in the .cadrc file. There is only one thing th remember about this cell: Its lower left corner must lie at the origin of its coordinate system. If this is not done, it will not be placed in the right locations. The defaule name of the cell is "contact".

7. The Log File

The log file, whose name is the circuit name with ".log" appendend, contains all the error, warning, and statistics messages that Termite and Yacr generate. Normally, the format will consist of some data about the number of cells and blocks, and then a section describing the routing of each channel.

A portion of a typical .log file is shown below:

There are 5 blocks.
313 cells were read in.

########################## ROUTING CHANNEL 1 ##########################

YACR 2.1: irregular channels and fixed end pins.

Input file: route.in.1

num_nets= 34, num_cols= 197, num_left_nets= 0, num_right_nets= 0
Columns with maximum density: 114 124 143 157 169
Best choice(s) for starting column is (are): 143 157
max density = 6
initial_column = 143
0 cycles were detected.
num_rows = 6

There are 80 vias
The total net_length is 827
The longest net is 1, metal length = 117, poly length = 9

6 rows were used.
1.4 real 0.8 user 0.2 sys
The channel width was adjusted by 1420.

Total length of geometry is 9633 lambda

Total channel width so far is: 6

########################## ROUTING CHANNEL 2 ##########################

YACR 2.1: irregular channels and fixed end pins.

Input file: route.in.2

num_nets= 62, num_cols= 210, num_left_nets= 0 num_right_nets= 0
Columns with maximum density: 117 147 161 165 167 175
Best choice(s) for starting column is (are): 165 167
max density = 9
initial_column = 165
0 cycles were detected.
num_rows = 9
VCV: nets 35 40, column 83
column 83 was routed by maze1a

There are 143 vias
The total net_length is 1600
The longest net is 29, metal length = 115, poly length = 8

9 rows were used.
4.1 real 1.2 user 0.3 sys

The top of the channel needs contact-jog clearance.
The bottom of the channel needs contact-jog clearance.
The channel width was adjusted by 60.

Total length of geometry is 28943 lambda

Total channel width so far is: 15

A happy ending!

The parts of the above example in **bold** type are produced directly by Termite. The first two lines say how many blocks and cells were read in. The number of cells includes the feedthroughs added by TimberWolf. Also, remember that there will be one more channel than the number of blocks.

This is followed by a section describing the routing of each channel. The parts of the example in regular type are produced by Yacr when it is automatically invoked to route the channels. The significance of these portions is explained in the *Yacr Users Guide*. It is sufficient to know that if Yacr has a problem that prevents it from correctly routing a channel, Termite will detect this. Also, the units and the net numbers used by Yacr are symbolic and not lambda or the actual net names.

The rest of each section gives statistics and trivia about the grid definition process and the amount of time and channel area used by the routing. Most of the dimensions will be in lambda, will some of the more obscure messages will have dimensions in Squid units (used internally by Termite). Messages describing fatal erors and internal errors will usually be in Squid units.

If a fatal error occurs, this fact will be noted in the log file, and the process will be terminated. Severe internal errors will cause a core dump to be created: save this for the maintainer of Termite to look at. The possible errors are described in the next section.

8. Errors in Termite

There are many potential fatal errors in Termite. However, if the input data is correct, only a few should ever appear.

If any of the tbird views of the cells in the circuit cannot be found by Squid, Termite will fail. Also, the bounding box of the cell (on the **SYMBOL** layer) must be present, as described in the *Flounder User's Guide*, and the above section on the contact and feedthrough cells.

The TimberWolf files read by Termite must have been produced by a run of TimberWolf done according to the instructions in *Notes on Running TimberWolf with ThunderBird*. It is assumed that the input to TimberWolf was prepared with the aid of Flounder. If not, the cell names and dimensions will not necessarily be in the correct format.

If the correct arrangement of blocks, pads, and macro cells is not used when TimberWolf is run, the global routing may not be in a form that Termite can deal with. In this case, Termite may give strange messages about nets not being routed properly, and the correct connections may not be made.

Missing or badly formatted lines in the .cadrc file will be fatal. It the design rules are bizarre, unforseen problems may arise. Termite has been tested with the mosisCMOS-PW design rules, and the layouts produced have passed design-rule checking by Lyra. Similar sets of rules should also work. One thing to beware of is making contacts narrower that the material they connect. This may make the grid spacing too tight to allow jogs to be made connecting the terminals to the gridded wiring in the channel.

The fatal error most likely to occur is that Termite will be unable to create a symbolic grid for the routing of some channel. This will happen if the terminals of the cells

are too close together, or if there is an unfortunate arrangement of the top and bottom terminals. The problem will be detected when Termite is unable to assign a terminal to a grid location. Various debugging messages will be placed in the log file, but Termite will continue by ignoring the offending pin. Having many consecutive terminals with the minimum spacing increases the chances of this happening. If this happens, the best cure is to check your cell designs to make sure the terminals are not too close together. Also, check what happens to the terminal spacing when various combinations of the cells are abutted. Adjusting the **GRID_WIDTH** parameter in the .cadrc file may help the grid definition to succeed, but design rule violations may be introduced as a result.

At this point, there are some messages that *might* indicate a problem. However, it is necessary to check the TimberWolf output files to make sure. The only known example is indicated by a line in the .log file of the form:

Net "*netname.nnn*": too few pins, net suppressed

This may occur if Timberwolf says that a net is connected to several places on one end of a channel, but nowhere else. This causes the routing file sent to YACR to have one net appearing in the left or right_list several times, but nowhere else. Because the net effectively has only one terminal, it is deleted, which is the right thing to do.

9. Running Termite Independently

It is possible to use Termite with the results of TimberWolf runs that were done independently of the ThunderBird package. In fact, Termite has been used on several industrial test cases which were created outside of the Hawk/Squid environment. At least two things must be done for this to work. First, library of Squid views of the cells must be created. Second, the units used for the coordinates must be converted.

To quickly create a dummy cell library, the probram **mkcell** can be used. This creates a tbird view of every cell used in the circuit. It is invoked as follows:

mkcell **circuitname**.pl1

Mkcell reads in the .pl1 file produced by TimberWolf, and creates a dummy cell consisting of a rectangle of metal and a rectangle of poly, as well as the rectangle marking the cell border. The feedthrough cell and the contact cell will have to be created manually.

If the units used for the cells were not half-lambda, the **INPUT_SCALE** parameter in the .cadrc file will have to be changed. The default value is 10, which converts from half-lambta to the Squid units used by TimberWolf. If this is changed, mkcell will have to be recompiled with the new value. See the source code (mkcell.c) to do this. Beware that if the circuit consists of many uniquely named cells, a separate Squid view will have to be created for each one. This will make Termite run very slowly, and possibly exceed the default maximum memory size. Use the **limit** command to increase your memory allocation. (See the manual entry for *csh(1)*.)